THE WALKING BUG 

Norme di progetto

The Walking Bug - 2019-04-29

Informazioni sul documento

Versione	4.0.0
Redazione	Gianmarco Santi
Verifica	Elisa Cattaneo
Responsabile	Marco Dugatto
Uso	Interno
Distribuzione	Prof. Tullio Vardanega Prof. Riccardo Cardin The Walking Bug
Stato	Approvato

Descrizione

Questo documento contiene tutte le regole che il gruppo TWB dovrà seguire per lo svolgimento di ogni attività, in qualsiasi ambito, per lo sviluppo del progetto.

Diario delle modifiche

Versione	Modifica	Autore	Ruolo	Data
4.0.0	*Approvazione del documento per il rilascio in RA*	Marco Dugatto	Responsabile	2019-04-29
3.1.0	*Verifica positiva del documento*	Elisa Cattaneo	Verificatore	2019-04-28
3.1.0	*Correzione errori evidenziati nella RQ*	Gianmarco Santi	Amministratore	2019-04-26
3.0.0	*Approvazione del documento per il rilascio in RQ*	Gianmarco Santi	Responsabile	2019-04-05
2.1.2	*Verifica positiva del documento*	Nicola Abbagnato, Graziano Grespan	Verificatore	2019-03-31
2.1.2	*Aggiunte metriche a §2.2.5.13 e §3.3.11.4*	Francesco De Filippis	Amministratore	2019-03-26
2.1.1	*Aggiunti strumenti a §2.2.6*	Francesco De Filippis	Amministratore	2019-03-25
2.1.0	*Verifica correzioni effettuate*	Elisa Cattaneo	Verificatore	2019-03-19
2.1.0	*Correzione errori evidenziati nella RP*	Francesco De Filippis	Amministratore	2019-03-18
2.0.0	*Approvazione per rilascio in RP*	Elisa Cattaneo	Responsabile	2019-02-04
1.3.0	*Verifica positiva del documento*	Gianmarco Santi, Graziano Grespan	Verificatore	2019-02-01
1.3.0	*Spostati Standard di processo, di prodotto e Ciclo di Deming in appendice*	Enrico Sanguin	Amministratore	2019-01-30
1.2.0	*Verifica documento*	Gianmarco Santi, Graziano Grespan	Verificatore	2019-01-28
1.2.0	*Aggiunta §2.2.4, §3.1.10, §3.5.5, §4.2.3*	Nicola Abbagnato	Amministratore	2019-01-24
1.1.2	*Aggiunta §2.2.3.5*	Enrico Sanguin	Amministratore	2019-01-23
1.1.1	*Aggiunta §2.2.2.2*	Enrico Sanguin	Amministratore	2019-01-23
1.1.0	*Correzione errori evidenziati nella RR*	Nicola Abbagnato, Enrico Sanguin	Amministratore	2019-01-22

1.0.0	*Approvazione per rilascio in RR*	Gianmarco Santi	Responsabile	2018-11-25
0.3.3	*Verifica positiva del documento*	Graziano Grespan, Francesco De Filippis	Verificatore	2018-11-25
0.3.3	*Correzioni errori §3 individuati in 0.3.0*	Enrico Sanguin	Amministratore	2018-11-24
0.3.2	*Correzioni errori sez 2 individuati in 0.3.0*	Nicola Abbagnato	Amministratore	2018-11-24
0.3.1	*Correzioni errori §4 individuati in 0.3.0*	Elisa Cattaneo	Amministratore	2018-11-24
0.3.0	*Verifica §4*	Francesco De Filippis	Verificatore	2018-11-23
0.3.0	*Verifica §3*	Graziano Grespan	Verificatore	2018-11-23
0.3.0	*Verifica §2*	Graziano Grespan	Verificatore	2018-11-22
0.3.0	*Prima stesura §4*	Elisa Cattaneo	Amministratore	2018-11-21
0.2.0	*Prima stesura §3*	Marco Dugatto	Amministratore	2018-11-21
0.1.0	*Prima stesura §2*	Nicola Abbagnato	Amministratore	2018-11-20
0.0.1	*Stesura scheletro del documento*	Marco Dugatto	Amministratore	2018-11-16

Indice

1 Introduzione

1.1 Scopo del documento

Questo documento è stato redatto al fine di garantire ai membri del gruppo TWB un *way of working*$_g$ comune, elencando le regole e le procedure da utilizzare per rendere la collaborazione fluida ed efficiente. Saranno inoltre descritti gli strumenti software che verranno utilizzati per rispettare le suddette regole.

1.2 Scopo del prodotto

Lo scopo del prodotto è la creazione di una *DApp*$_g$ su rete *Ethereum*$_g$ che sia accessibile tramite *MetaMask*$_g$ da *Mozilla Firefox*$_g$ 9.1 e *Google Chrome*$_g$ 71 attraverso un'*interfaccia web*$_g$.
La *DApp*$_g$ deve prevedere tre tipi di utenti con le relative funzionalità:

- **Stato**:
  - Creare *Cubit*$_g$ e distribuirli;
  - Inviare la richiesta di pagamento delle tasse e verificarne lo stato;
  - Gestire la lista delle attività registrate.
- **Detentori di partita IVA**:
  - Registrare la propria attività alla lista dello Stato;
  - Gestire i propri beni e servizi sul mercato;
  - Fare compravendita di beni e servizi da altre attività;
  - Gestire le tasse ricevute dallo stato.
- **Cittadini**:
  - Comprare servizi e beni dalle attività.

1.3 Glossario

Nel *Glossario* vengono inseriti tutti quei termini considerati potenzialmente ambigui o poco chiari così da darne una spiegazione semplice e concisa. Tutti i vocaboli che si trovano nel glossario sono scritti nei diversi documenti in corsivo e con una $_g$ a pedice.

1.4 Riferimenti utili

1.4.1 Riferimenti normativi

- **Riferimenti organigramma del corso di Ingegneria del Software**: https://www.math.unipd.it/~tullio/IS-1/2018/Progetto/RO.html.

1.4.2 Riferimenti informativi

- Standard [ISO/IEC 15504]: https://en.wikipedia.org/wiki/ISO/IEC_15504;
- Standard [ISO/IEC 25010];
- **Corso di Ingegneria del Software - Analisi dei Requisiti**: http://www.math.unipd.it/~tullio/IS-1/2018/Dispense/L08.pdf;
- **Airbnb JavaScript Style Guide**: https://github.com/airbnb/javascript;
- **Airbnb React/JSX Style Guide** : https://github.com/airbnb/javascript/tree/master/react;
- **Airbnb CSS/Sass Style Guide**: https://github.com/airbnb/css.

2 Processi primari

2.1 Fornitura

2.1.1 Scopo

Questo capitolo presenta le norme in ambito di progettazione, sviluppo e consegna del prodotto Soldino che il gruppo *The Walking Bug*$_g$ vara al fine di proporsi e diventare fornitore alla proponente *Red Babel*$_g$ e ai *committenti*$_g$ Prof. Tullio Vardanega e Prof. Riccardo Cardin.

2.1.2 Aspettative

Come suggerito dalla proponente *Red Babel*$_g$ il gruppo si impegna a mantenere un colloquio costante, al fine di sanare qualsiasi tipo di dubbio su requisiti di vincolo e di qualità richiesti per il prodotto, ed arrivare così ad un accordo finale tra le due parti che definisca il contratto a cui attenersi.

2.1.3 Attività

2.1.3.1 Studio di fattibilità

In seguito alla presentazione dei Capitolati d'appalto, avvenuta venerdì 16 Novembre nell'aula 1C150 di Torre Archimede, è stata convocata una riunione ufficiale per la discussione delle varie proposte presentate, evidenziandone per ognuna i punti di interesse comuni ai membri del gruppo.
In seguito alla scelta del capitolato, per il quale proporsi come fornitori, gli analisti hanno svolto un'ulteriore attività di analisi dei rischi, stilando così il documento *Studio di fattibilità v1.0.0*. Questo documento presenta brevemente tutti i capitolati esposti dalle proponenti e indica le motivazioni che hanno portato il gruppo a proporsi come fornitore per il prodotto scelto. Il documento, per ogni *capitolato*$_g$, è articolato come segue:

- **Descrizione**: breve descrizione del prodotto da sviluppare;
- **Studio del dominio**: analisi del target cui si rivolge il prodotto richiesto;
- **Valutazione**: esposizione dei principali aspetti positivi e negativi che hanno influenzato la scelta;
- **Conclusioni**: motivazione dettagliata della decisione compiuta dal gruppo.

2.1.3.2 Piano di Progetto

Il Responsabile di Progetto, con il supporto degli Amministratori, dovrà individuare una strategia da attuare per la realizzazione del progetto, cercando di ottimizare l'utilizzo delle risorse a disposizione. Il documento sarà composto dai seguenti contenuti:

- **Analisi dei Rischi**: l'analisi dei rischi permette di analizzare i rischi che potrebbero insorgere durante lo sviluppo del progetto. L'obiettivo è quello di anticipare i rischi che potrebbero influire sulla pianificazione del progetto o sulla qualità del software ed eventulamente prevedere un modo per fare in modo che il loro impatto sia minimo;
- **Modello di Sviluppo**: scelta del modello di sviluppo che il gruppo adotterà;

- **Pianificazione**: vengono pianificate le attività da svolgere, corredate da scadenze temporali precise;
- **Preventivo e Consuntivo**: sulla base della pianificazione effettuata, si realizza un preventivo contenente il costo totale del progetto. Al termine di ciascuna attvità viene realizzato un consuntivo di periodo utile per capire l'andamento dei costi rispetto a quanto preventivato.

2.1.3.3 Piano di Qualifica

I Verificatori si occuperanno di scegliere la modalità più appropriata per svolgere l'attività di verifica e di validazione del materiale prodotto dal team. Il documento sarà composto dai seguenti contenuti:

- **Obiettivi di qualità**: individuazione degli standard per accertare la qualità di prodotto e la qualità di processo. Vengono inoltre specificati i valori di soglia delle metriche individuate e descritte nel documento *Norme di Progetto_ v3.0.0*;
- **Specifiche dei test**: definizione dettagliata delle tipologie di test che verranno effettuati sul prodotto;
- **Resoconto delle attività di verifica**: alla fine di ciascuna attività vengono istanziate delle metriche con i relativi valori calcolati, i quali verranno rappresentati tramite serie storiche e diagrammi.

2.1.3.4 Collaudo e Consegna del Prodotto

Prima della consegna, il gruppo sottoporrà il prodotto al collaudo, il quale si terrà in presenza del committente e della proponente. Prima di arrivare a quest'ultima fase, il gruppo dovrà assicurarsi della correttezza, completezza e affidabilità del software sviluppato. In particolare, durante il collaudo, dovranno essere dimostrati i seguenti punti:

- L'esecuzione di tutti i test di validazione descritti nel documento *Piano di Qualifica_ v3.0.0*;
- Il soddisfacimento di tutti i requisiti obbligatori descritti nel documento *Analisi dei Requisiti_ v3.0.0*;
- Il soddisfacimento di alcuni dei requisiti desiderabili e opzionali descritti nel documento *Analisi dei Requisiti_ v3.0.0.*

Una volta consegnato il prodotto finale il gruppo *The Walking Bug*$_g$ sarà menzionato nei diritti d'autore dell'applicazione; inoltre, salvo diversi accordi con la proponente, il gruppo non si prenderà in carico di effettuare eventuali attività di manutenzione del prodotto.

2.2 Sviluppo

Questa sezione raccoglie le linee guida usate durante i processi di *analisi*$_g$, *progettazione*$_g$ e codifica del prodotto; inoltre presenta gli strumenti utilizzati dal gruppo per compiere tali attività.

2.2.1 Analisi dei requisiti

L'analisi dei Requisiti è un documento che elenca tutti i requisiti del *capitolato*$_{\text{g}}$ e le fonti da cui sono stati ricavati:

- *Capitolato*$_{\text{g}}$;
- Verbali degli incontri esterni;
- *Casi d'uso*$_{\text{g}}$.

L'obiettivo di tale documento è stabilire in modo dettagliato e accurato gli obiettivi che andranno perseguiti durante lo sviluppo del prodotto.

2.2.1.1 Classificazione dei requisiti

Ad ogni requisito verrà associato un codice univoco in questo formato:

R[Importanza][Categoria][Codice]

- **R:** requisito;
- **Importanza:** un valore tra:
  - **2**: requisito obbligatorio, necessario affinché il prodotto garantisca almeno le funzioni di base;
  - **1**: requisito desiderabile, non pregiudica le funzioni di base ma porta alla completezza del prodotto finale;
  - **0**: requisito opzionale, non influenza il prodotto a livello funzionale.
- **Categoria:** un valore tra:
  - **F**: requisito funzionale, descrive il comportamento che il software deve avere in particolari situazioni;
  - **Q**: requisito di qualità, descrive dei parametri che garantiscono un prodotto di qualità;
  - **V**: requisito di vincolo, descrive un vincolo posto dalla proponente.
  - **P**: requisito prestazionale, descrive dei vincoli sulle prestazioni che il prodotto dovrà garantire.
- **Codice:** un numero progressivo che permette di riconoscere univocamente il requisito.

2.2.1.2 Classificazione dei casi d'uso

Come i requisiti, anche i *casi d'uso*$_{\text{g}}$ vanno catalogati assegnando ad ognuno di essi un identificatore univoco in questo formato:

UC[Codice]

- **UC:** *Use Case*$_{\text{g}}$;
- **Codice:** è una serie univoca di cifre divise tra loro da '.' in modo da poter dividere in modo gerarchico i *casi d'uso*$_{\text{g}}$.

Oltre all'identificatore ogni *caso d'uso*$_g$ viene descritto dalle seguenti informazioni:

- **Nome:** il nome associato al *caso d'uso*$_g$;
- **Descrizione:** breve descrizione della situazione modellata;
- **Attori:** gli attori sia primari sia secondari coinvolti;
- **Scenario principale:** elenco numerato degli eventi che avvengono nel *caso d'uso*$_g$;
- **Precondizione:** condizioni che si assumono essere vere prima degli eventi descritti nello scenario principale;
- **Postcondizione:** condizioni che si suppongono vere dopo degli eventi descritti nello scenario principale;
- **Inclusioni:** *casi d'uso*$_g$, se presenti, che si verificano per intero tra gli eventi descritti nello scenario principale;
- **Estensioni:** *casi d'uso*$_g$, se presenti, che figurano questo *caso d'uso*$_g$ all'interno dello scenario principale.

2.2.1.3 Diagrammi dei casi d'uso

In questi diagrammi ogni attore deve essere rappresentato da un uomo stilizzato accompagnato dal suo nome.

Figura 1: Attore in un caso d'uso

Ogni caso d'uso deve essere rappresentato da un ovale al cui interno è presente l'identificativo del caso d'uso (e.g. UCx) accompagnato da un nome descrittivo. Ogni attore deve essere associato ad almeno un caso d'uso e viceversa, tramite una linea semplice. Nel caso in cui siano presenti delle estensioni per un caso d'uso, queste devono essere indicate all'interno dello stesso in una sezione separata orizzontalmente dall'identificativo.

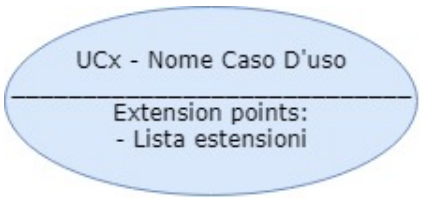

Figura 2: Caso d'uso con estensioni

Le estensioni devono essere indicate con la notazione `«extend»` e collegate al caso d'uso da cui estendono con una freccia tratteggiata; inoltre le specifiche per l'avvenimento di una

particolare estensione devono essere rappresentate da un foglio con un angolo piegato, al cui interno ci saranno le informazioni, collegato con una linea tratteggiata alla freccia.

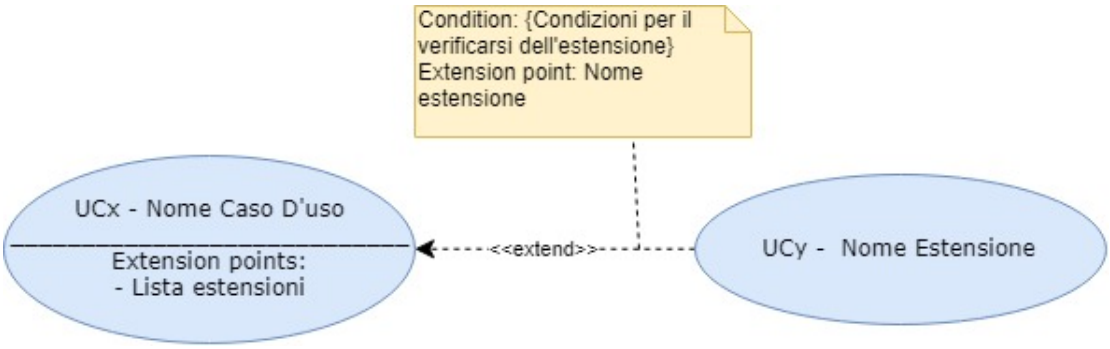

Figura 3: Estensione in un caso d'uso

Le inclusioni devono essere indicate con la notazione `«include»` e collegate al caso d'uso da cui estendono con una freccia tratteggiata.

Figura 4: Inclusione in un caso d'uso

Le generalizzazioni devono essere rappresentate da una freccia vuota, e possono collegare degli attori così come dei casi d'uso.

Figura 5: Generalizzazione in un caso d'uso

Lo strumento utilizzato per lo sviluppo di questi diagrammi è *draw.io*$_g$[1]. Trattandosi di un'applicazione web, ogni diagramma completato dovrà essere esportato come file *jpeg* in scala *100%* e accompagnato dal file *xml* sorgente, al fine di essere facilmente manutenibile e modificabile in futuro. Lo strumento appena descritto verrà inoltre utilizzato per lo sviluppo di tutte le tipologie di diagrammi descritti nella sezione 2.2.2.3.

2.2.2 Progettazione

2.2.2.1 Scopo

L'attività di *progettazione*$_g$ ha come scopo la creazione dell'*architettura*$_g$ che il prodotto finale dovrà seguire. Questa dovrà essere costruita in modo tale che:

- Sia capace di soddisfare tutti i requisiti;
- Sia facilmente modificabile nel caso cambiassero i requisiti;
- Sia facilmente comprensibile dagli *stakeholder*$_g$;
- "Resista" all'errore umano e ad interferenze da parte dell'ambiente;
- Sia efficiente, sia nello spazio che nel tempo;
- Richieda poco tempo per essere manutenuta;
- Non vi siano gravi malfunzionamenti;
- Non sia vulnerabile ad *intrusioni*$_g$;
- Solo l'esterno sia visibile agli utenti;
- Sia composta da parti distinte e indipendenti tra loro, meglio ancora se riusabili in future applicazioni;
- Le *componenti*$_g$ siano divise in base al loro obiettivo;
- Non abbia componenti superflui.

2.2.2.2 Integrazione

Lo sviluppo dell'applicazione Soldino sarà accompagnato da un'attività di integrazione continua ed automatizzata, che verrà effettuata tramite il servizio *Travis CI*$_g$ compatibile con *Github*$_g$.
Con questo strumento, ad ogni commit verrà creata una build del prodotto ed eseguiti i test di unità, in modo da identificare facilmente le modifiche o gli eventuali bug presenti.

2.2.2.3 Diagrammi

Nella fase di progettazione verrà fatto largo uso di *Diagrammi UML2.0*$_g$, in modo da evitare eventuali ambiguità riguardo le scelte progettuali e renderle chiare a tutti i membri del gruppo.
I diagrammi da utilizzare sono:

[1] https://www.draw.io/

- **Diagrammi di attività**: per descrivere la logica procedurale e gli aspetti dinamici dei casi d'uso;
- **Diagrammi delle classi**: per descrivere il tipo di oggetti che fanno parte del sistema e le loro dipendenze;
- **Diagrammi di sequenza**: per descrivere il comportamento collaborativo che un gruppo di oggetti deve implementare;
- **Diagrammi dei package**: per descrivere il raggruppamento di un gruppo di elementi UML in un'unità di livello più alto.

2.2.2.3.1 Diagrammi di attività

I diagrammi di attività devono contenere i seguenti elementi, tutti collegati da frecce semplici e rappresentati come segue:

- **Nodo iniziale**: rappresentato da un pallino nero pieno, è il punto da cui inizia l'esecuzione del caso d'uso e genera un token;
- **Attività**: rappresentata da un rettangolo con gli angoli arrotondati che ne contiene un breve nome descrittivo. Consuma e genera un token;
- **Sotto-attività**: rappresentata allo stesso modo di un'attività ma con un piccolo tridente nell'angolo in basso a sinistra, per indicare che si tratta di un riferimento ad una sotto-attività descritta in un diagramma a parte. Ogni sotto-attività ha un input e un output;
- **Fork**: rappresentato da una linea nera orizzontale o verticale, è un punto in cui l'attività si parallelizza. Consuma un token e ne genera quante sono le frecce in uscita;

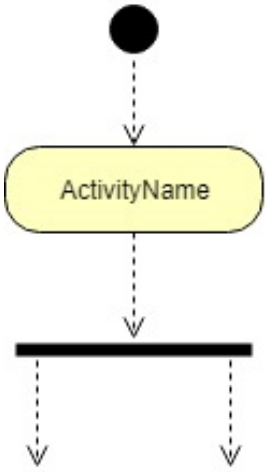

Figura 6: Nodo iniziale, Attività e Fork in un diagramma di attività

- **Join**: rappresentata da una linea nera orizzontale o verticale, è un punto in cui avviene la sincronizzazione tra processi paralleli iniziata in una *Fork*. Ha n frecce in entrata e una in uscita, consuma n token e ne genera uno;

- **Branch**: rappresentato da un rombo, è un punto in cui si può intraprendere solo uno degli n rami/frecce in uscita, ognuno dei quali deve riportare la condizione/guardia per il quale può essere intrapreso. Consuma un token e ne genera un altro;
- **Merge**: rappresentato da un rombo con n frecce in entrata e una in uscita, è un punto in cui gli n percorsi generati da un *Branch* si riuniscono. Consuma un token e ne genera un altro;

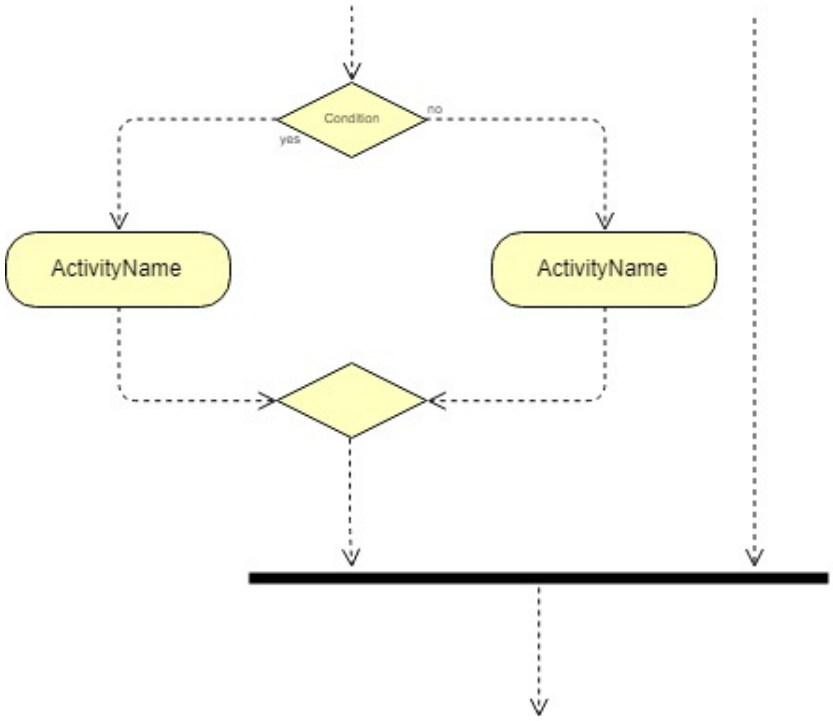

Figura 7: Branch, Merge e Join in un diagramma di attività

- **Pin**: rappresentato da un piccolo quadrato dove entrano o escono frecce dalle *Attività*, indica il passaggio di un parametro, il cui tipo va indicato a fianco;
- **Segnali**: rappresentati da due figure ad incastro, la prima per l'invio (non bloccante) e la seconda per la ricezione dello stesso (bloccante);

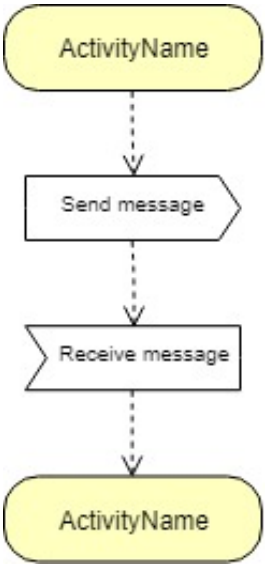

Figura 8: Segnali in un diagramma di attività

- **Timeout**: rappresentato da una clessidra, indica un evento ripetuto o appunto un timeout, e può avere frecce entranti e uscenti. Devono essere accompagnati da una descrizione in formato `wait indicazioneTemporale` per un timeout o `every indicazioneTemporale` per gli eventi ripetuti;
- **Nodo di fine flusso**: rappresentato da un cerchio con una 'X' al suo interno, indica il punto in cui uno dei rami di esecuzione va a morire e consuma un token. L'esecuzione continua comunque sugli altri rami paralleli;
- **Nodo finale**: rappresentato da due cerchi concentrici in cui il più interno è pieno, è il punto in cui termina l'esecuzione e consuma un token.

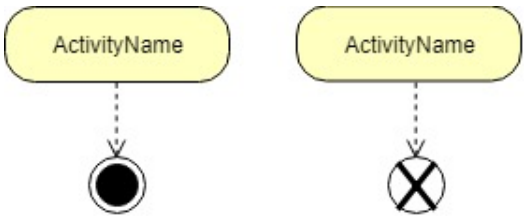

Figura 9: Nodi finale e di fine flusso in un diagramma di attività

2.2.2.3.2 Diagrammi delle classi

React
Le componenti React sono rappresentate da rettangoli suddivisi in tre sezioni nel seguente modo:

- **Nome componente**: deve essere indicato in lingua inglese, seguire la notazione *camelCase* ed essere univoco;

- **Props**: devono essere indicati secondo la seguente notazione:
  ```
  propName :  type
  ```
  nel caso in cui un prop non abbia componenti questa sezione viene lasciata vuota;
- **Methods**: lista dei metodi del componente;
- **Render**: descrizione del componente.

Redux
Le componenti Redux sono le *Actions* e i *Reducers*, entrambe rappresentate da rettangoli.
Le Actions contengono:

- **type**: è indicato da una stringa e contiene un nome descrittivo della funzionalità dell'azione. Deve essere scritto tutto in maiuscolo;
- **payload**: è il dato che viene effettivamente passato dall'azione al reducer.

I Reducers invece contengono:

- **initialState**: lo stato iniziale del reducer, rappresentato tipicamente da un $JSON_{\text{g}}$ contenente il tipo dell'azione ricevuta e il suo payload;
- **actionCheck**: uno statement che controlla il tipo di un'azione e cambia lo stato dello store in base al tipo e al payload di questa azione.

Solidity
I contratti in Solidity sono rappresentati da rettangoli suddivisi in tre sezioni nel seguente modo:

- **Nome contratto**: deve essere indicato in lingua inglese, seguire la notazione *camelCase* ed essere univoco;
- **Campi dati**: lista dei campi dati del contratto, che devono essere indicati secondo la seguente notazione:
  ```
  visibility name :  type
  ```
  dove *visibility* deve assumere uno dei seguenti valori:
  - '+': indica la visibilità pubblica;
  - '-': indica la visibilità privata;
  - '#': indica la visibilità interna/protetta.
- **Modifiers**: lista dei modificatori del contratto, che devono essere indicati secondo la seguente notazione:
  ```
  modifierName (parameterName :  parameterType)
  ```
- **Metodi**: lista dei metodi del contratto, che devono essere indicati secondo la seguente notazione:

Per ogni linguaggio utilizzato ogni classe implementata apparirà nel diagramma, anche se non dovesse possedere metodi e/o attributi propri. Le classi in questi diagrammi sono collegate tra loro da delle frecce che ne indicano il tipo di dipendenza. In particolare:

- Una freccia semplice dalla classe A alla classe B indica una dipendenza forte, ovvero che A ha fra i propri campi dati una o più istanze della classe B;

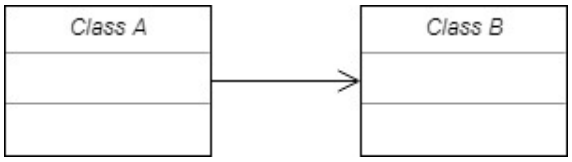

Figura 10: Relazione di dipendenza forte

- Una freccia tratteggiata dalla classe A alla classe B indica una dipendenza debole, ovvero che A ha interagisce con la classe B secondo una primitiva, segnata come sotto;

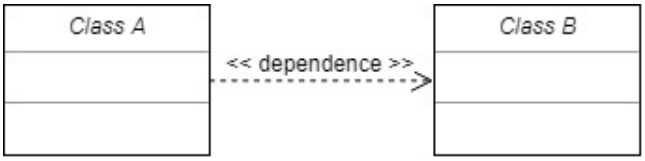

Figura 11: Relazione di dipendenza debole

- Una freccia a diamante vuoto dalla classe A alla classe B indica un'aggregazione, ovvero che la classe B appartiene ad A;

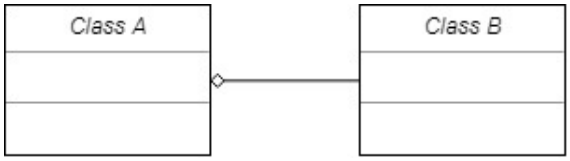

Figura 12: Relazione di Aggregazione

- Una freccia a diamante pieno dalla classe A alla classe B indica una composizione, ovvero che la classe B appartiene solamente ad A e può essere istanziata solo da questa;

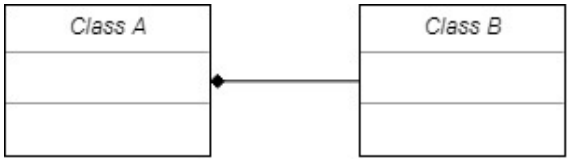

Figura 13: Relazione di composizione

- Una freccia vuota dalla classe A alla classe B indica una relazione di generalizzazione/ereditarietà, ovvero che ogni istanza di A è anche un'istanza di B;

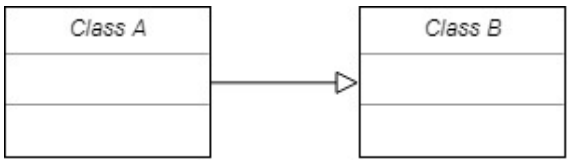

Figura 14: Relazione di generalizzazione/ereditarietà

2.2.2.3.3 Diagrammi di sequenza

Nei diagrammi di sequenza il senso di lettura sarà dall'alto verso il basso, ad indicare lo scorrere del tempo. Tutti gli oggetti di un diagramma di sequenza devono essere rappresentati da un rettangolo, al cui interno si deve trovare il nome per identificarli nel formato *acronimo : NomeClasse*. Al di sotto di ogni oggetto deve essere rappresentata la sua linea della vita tratteggiata: lungo questa linea ci potranno essere delle barre di attivazione per ogni oggetto, che staranno ad indicare i momenti in cui l'oggetto è attivo. Dalle barre di attivazione potranno partire delle frecce, verso le linee della vita di altri oggetti già istanziati o verso nuovi oggetti da istanziare.
Queste frecce devono essere rappresentate in uno di questi formati:

- **Freccia piena**: indica un messaggio sincrono, ovvero l'invocazione di un metodo con attesa del `return`. Sopra a queste frecce deve essere indicato il nome del metodo invocato secondo il formato:

  `nomeMetodo(listaParametri)`

- **Freccia semplice**: indica un messaggio asincrono, ovvero l'invocazione di un metodo senza l'attesa del `return` corrispondente;
- **Freccia tratteggiata**: indica un messaggio di `return` di un metodo invocato. Sopra a queste frecce deve essere indicato il tipo del dato di ritorno, secondo il formato `tipoRitorno`;
- **Freccia tratteggiata** `«create»`: indica la creazione di una nuova istanza di un oggetto e deve sempre terminare in un rettangolo rappresentante il nuovo oggetto;
- **Freccia tratteggiata** `«destroy»`: indica la distruzione di un'istanza di un oggetto e deve sempre terminare in una 'X', con la quale finirà anche la linea della vita dell'oggetto.

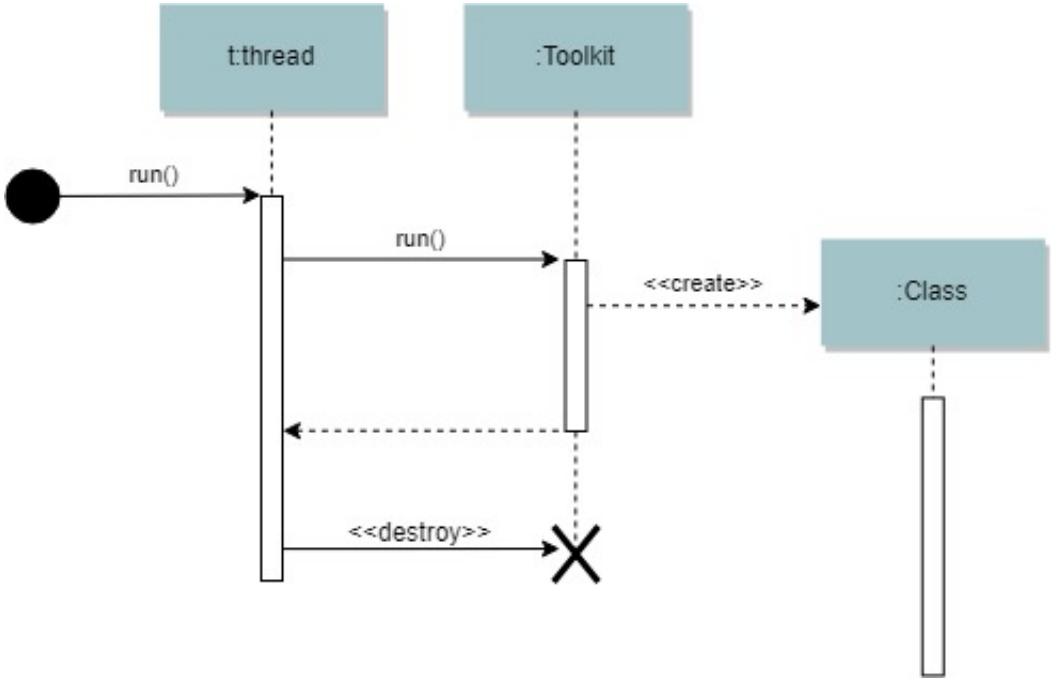

Figura 15: Classi e relazioni nei diagrammi di sequenza

Potranno inoltre essere riportati dei *frame di interazione*, associati a delle condizioni, in particolare:

- **alt**: indica dei frammenti multipli in alternativa, dei quali verrà eseguito solo quello per cui è verificata la condizione;
- **opt**: indica un frammento che viene eseguito solo se la condizione specificata è verificata. Equivalente ad *alt* con solo una freccia;
- **par**: indica dei frammenti eseguiti in parallelo;
- **loop**: indica un frammento che può essere eseguito più volte, in cui la base dell'iterazione è indicata dalla condizione;
- **region**: indica una sezione critica in cui il frammento può essere eseguito da un solo thread alla volta;
- **neg**: indica un frammento con un'interazione non valida;
- **ref**: indica un riferimento ad un'interazione definita in un altro diagramma;
- **sd**: utilizzato per racchiudere un intero diagramma di sequenza.

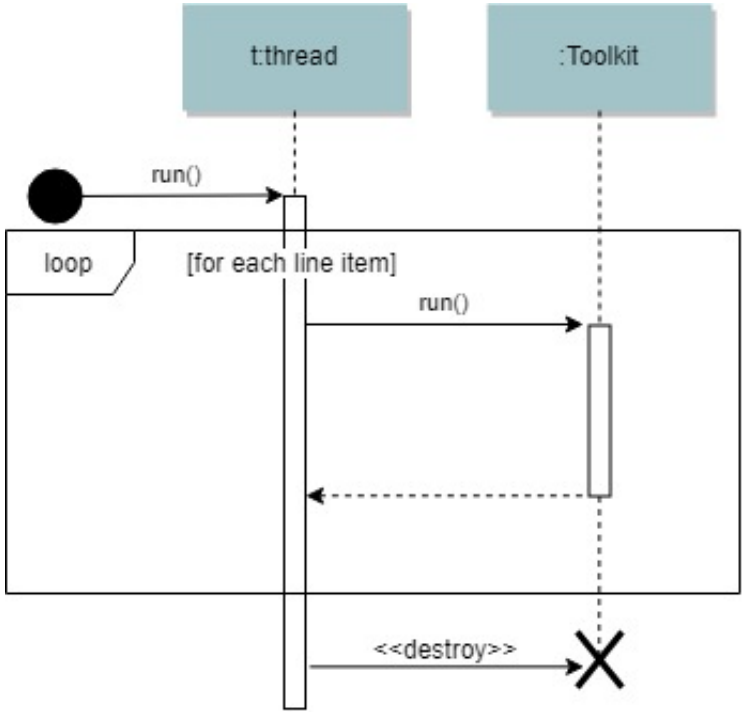

Figura 16: Frame di interazione nei diagrammi di sequenza

2.2.2.3.4 Diagrammi dei package

Ogni package deve essere rappresentato da un rettangolo con un'etichetta per il nome, e al suo interno verranno riportate le classi che ne fanno parte, così come degli eventuali sotto-package. Le dipendenze tra i vari package dovranno essere rappresentate mediante una freccia tratteggiata.

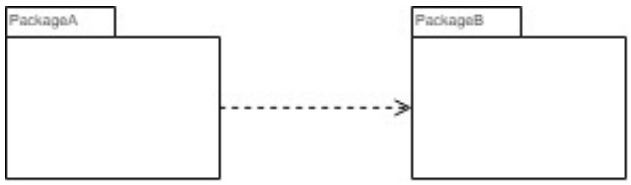

Figura 17: Dipendenza tra package

Come esempio, nella figura sopra la freccia indica una dipendenza di *PackageA* nei confronti di *PackageB*.

2.2.2.4 Obiettivi della progettazione

La progettazione garantisce che il prodotto soddisfi la qualità e i requisiti, definiti nell'attività di analisi, tramite:

- Correttezza per costruzione, in modo da adottare a pieno il modello di sviluppo incrementale;
- Una corretta suddivisione delle risorse;

- Una corretta organizzazione e ripartizione delle attività di implementazione, in modo da rendere facilmente risolvibili i problemi nell'attività di codifica da parte dei Programmatori;
- Un'efficace attività di implementazione architetturale, in modo da renderla chiara e comprensibile;
- Modularità, in modo da mantenere una bassa dipendenza tra le varie componenti del prodotto.

2.2.3 Strumenti di sviluppo

2.2.4 Convenzioni di codifica

Le convenzioni di codifica forniscono delle *best practice*$_{\text{g}}$ per lo stile di programmazione che il team deve adottare.
Esse comprendono:

- Regole di dichiarazione e denominazione per variabili e funzioni;
- Regole per l'uso di spazi bianchi, indentazione e commenti;
- Prassi e principi di programmazione.

Le convenzioni di codifica permettono inoltre di:

- Migliorare la leggibilità del codice;
- Rendere agevole la manutenzione del codice sorgente.

L'uso di norme e convenzioni di codifica è essenziale per garantire la qualità del prodotto finale.

In riferimento ai linguaggi di programmazione richiesti per lo sviluppo del progetto, i Programmatori devono attenersi alle norme di codifica indicate di seguito.

2.2.4.1 Convenzione di codifica JavaScript

2.2.4.1.1 Airbnb JavaScript Style Guide

In questo paragrafo vengono elencate alcune delle norme tratte da *Airbnb JavaScript Style Guide*[2].

Durante lo sviluppo del software il team si impegna comunque a rispettare tutte le norme inserite nella guida di stile ufficiale Airbnb.

Il controllo delle norme è applicato automaticamente attraverso il tool *ESlint*$_g$[3], configurato sulla base della guida di stile redatta dalla community di Airbnb[4] e utilizzato assieme ad uno degli IDE presenti nella guida per la configurazione.

La versione dello standard *ECMA*$_g$[5] richiesta dal committente e seguita dal team è la 2017 detta anche *ES8*.

Di seguito alcune delle norme risultate ai membri del gruppo significative e degne di menzione:

- Utilizzare sempre la keyword `const` per le variabili. Nel caso in cui una variabile deve essere riassegnata, va utilizzata la keyword *let* e non *var*;

```
1  // bad
2  var a = 1;
3  var b = 2;
4
5  // good
6  const a = 1;
7  const b = 2;
```

- Utilizzare sempre i singoli apici per le stringhe;

```
1  // bad
2  const name = "Capt. Janeway";
3
4  // bad - template literals should contain interpolation or ↩
       newlines
5  const name = `Capt. Janeway`;
6
7  // good
8  const name = 'Capt. Janeway';
```

- Per la creazione programmatica di stringhe utilizzare sempre i template e non la concatenazione;

[2] https://github.com/airbnb/javascript
[3] https://eslint.org/
[4] https://github.com/doasync/eslint-config-airbnb-standard
[5] https://en.wikipedia.org/wiki/ECMAScript

```
1  // bad
2  function sayHi(name) {
3    return 'How are you, ' + name + '?';
4  }
5
6  // bad
7  function sayHi(name) {
8    return ['How are you, ', name, '?'].join();
9  }
10
11 // bad
12 function sayHi(name) {
13   return `How are you, ${ name }?`;
14 }
15
16 // good
17 function sayHi(name) {
18   return `How are you, ${name}?`;
19 }
```

- Utilizzare la sintassi apposita per i parametri di default.

```
1   // really bad
2   function handleThings(opts) {
3   // No, We shouldn't mutate function arguments.
4   // Double bad: if opts is falsy it'll be set to an object ↩
        which may
5   // be what you want but it can introduce subtle bugs.
6   opts = opts || {};
7   // ...
8   }
9
10  // still bad
11  function handleThings(opts) {
12  if (opts === void 0) {
13  opts = {};
14  }
15  // ...
16  }
17
18  // good
19  function handleThings(opts = {}) {
20  // ...
21  }
```

2.2.4.1.2 Altre convenzioni adottate

- **Nomi delle variabili**: lo stile *camelCase*[6] deve essere utilizzato per i nomi di identificatori (variabili e funzioni). Tutti i nomi devono inoltre iniziare con una lettera;

[6] https://en.wikipedia.org/wiki/Camel_case

```
1     var firstName = "John";
2     var lastName = "Doe";
3
4     var price = 19.90;
5     var tax = 0.20;
6
7     var fullPrice = price + (price * tax);
```

- **Spaziatura intorno agli operatori**: vanno sempre inserite spaziature intorno agli operatori: `= + - * / ,` e dopo le virgole;

```
1     var x = y + z;
2     var values = ["Volvo", "Saab", "Fiat"];
```

- **Indentazione**: vanno sempre usati 4 spazi per l'indentazione dei blocchi di codice;

```
1     function toCelsius(fahrenheit) {
2         return (5 / 9) * (fahrenheit - 32);
3     }
```

- **Regole di dichiarazione**:
  - Terminare sempre ogni statement con un punto e virgola;
  - Inserire la parentesi graffa di apertura all'estremità della prima riga;
  - Utilizzare uno spazio prima della parentesi di apertura;
  - Inserire la parentesi di chiusura su una nuova linea, senza spazi iniziali;
  - Posizionare la parentesi di apertura sulla stessa linea, come il nome dell'oggetto;
  - Utilizzare i due punti ed uno spazio tra ogni proprietà e il suo valore;
  - Usare le virgolette intorno ai valori di stringa, non intorno a valori numerici;
  - Non aggiungere una virgola dopo l'ultima coppia proprietà-valore;
  - Terminare sempre una definizione di un oggetto con il punto e virgola.

```
1     var values = ["Volvo", "Saab", "Fiat"];
2
3     var person = {
4       firstName: "John",
5       lastName: "Doe",
6       age: 50,
```

```
7       eyeColor: "blue"
8     };
9
10    if (time < 20) {
11      greeting = "Good day";
12    } else {
13      greeting = "Good evening";
14    }
```

- **Lunghezza linea <80**: per facilitare la lettura, evitare linee più lunghe di 80 caratteri.

  Se una dichiarazione *JavaScript*$_g$ non si adatta su una sola riga, il posto migliore per andare a capo, è dopo un operatore o una virgola.

```
1     document.getElementById("demo").innerHTML =
2     "Hello Dolly.";
```

2.2.4.1.3 Best practice

- Evitare le variabili globali;
- Inizializzare sempre le variabili;
- Non utilizzare la keyword `new Object()`;
- Utilizzare `===` per il confronto.

2.2.4.2 Convenzione di codifica CSS / Sass

La convenzione di codifica è tratta dalla guida di stile *Airbnb CSS / Sass Styleguide*[7].

- **BEM**, o "Block-Element-Modifier"[8][9]:
  - Si consiglia di utilizzare la convenzione di nomenclatura BEM che segue questo pattern:

```
1       .block {}
2       .block__element {}
3       .block--modifier {}
```

    * **.block** è un blocco, rappresenta il livello più alto di un astrazione o un componente (la radice);
    * **.block__element** è un figlio di .block, rappresenta un elemento discendente da .block che lo compone;
    * **.block–modifier** è un modificatore, rappresenta una versione differente di .block.

[7] https://github.com/airbnb/css
[8] https://css-tricks.com/bem-101
[9] http://csswizardry.com/2013/01/mindbemding-getting-your-head-round-bem-syntax/

  - In particolare si consiglia una variante della BEM dove i blocchi vengono scritti con il *PascalCase*$_g$, questa funziona particolarmente bene quando combinata con *React*$_g$. Vengono ancora utilizzati i trattini bassi `__` e i trattini alti `--` per figli e modificatori.

- **Formattazione**:
  - Usare 2 spaziature per l'indentazione;
  - Per i nomi di classe preferire il trattino alto all'uso di camelCase (ad eccezione dei blocchi nella BEM);
  - Non utilizzare i selettori ID;
  - Quando si usano selettori multipli in una regola di dichiarazione, ogni selettore deve avere avere la propria riga;
  - Inserire una linea vuota tra ogni nuova regola;
  - Sono da evitare i selettori annidati, se sono però strettamente necessari si può arrivare al massimo a 3 livelli di annidamento.

```
1   .avatar {
2     border-radius: 50%;
3     border: 2px solid white;
4   }
5
6   .one,
7   .selector,
8   .per-line {
9     // ...
10  }
```

- **Commenti:**
  - I commenti su una singola linea `//` sono da preferire rispetto a quelli di blocco `/* */`;
  - È preferibile che i commenti abbiano linee proprie;
  - Sono da evitare i commenti a fine linea.
- **Hook Javascript:**
  - È da evitare l'utilizzo delle stesse classi in CSS e JavaScript;
  - È raccomandato l'utilizzo di classi specifiche per JavaScript, con il prefisso `.-js`.
- **Sintassi**: va usata la sintassi .scss e non l'originale .sass;

- **Ordine di dichiarazione delle proprietà**:
  - Si parte con le dichiarazioni standard;
  - Vanno raggruppati gli `@include` alla fine della dichiarazione;
  - I selettori annidati, se necessari, vanno posti dopo tutte le proprietà.

```
1  .btn {
2    background: green;
3    font-weight: bold;
4    @include transition(background 0.5s ease);
5
6    .icon {
7      margin-right: 10px;
8    }
9  }
```

- **Variabili**:
  - È da preferire usare i trattini medi nei nomi delle variabili piuttosto che il *camelCase*$_g$;
  - È accettabile precedere i nomi delle variabili utilizzate solo all'interno dello stesso file con un trattino basso (e.g. `$_la-mia-variabile`).
- **Mixin**: i mixin devono essere usate per semplificare e accorciare il codice;
- **Direttiva extend**: la direttiva `@extend` va evitata in quanto ha un comportamento controintuitivo e potenzialmente pericoloso.

2.2.4.3 Convenzione di codifica Solidity

In questa sezione vengono indicate le norme tratte dalla documentazione ufficiale di Solidity[10]; il codice verrà scritto in un editor che supporti determinati plugin[11,12], opportunamente configurati.

- **Indentazione:**
  - Le spaziature sono da preferire come metodo di indentazione;
  - Non è consentito l'utilizzo promiscuo di tabulazioni e spazi;
  - Vanno utilizzate 4 spaziature per ogni livello di indentazione.
- **Righe vuote:**
  - Nel codice sorgente le dichiarazioni di primo livello vanno separate da 2 righe vuote;

[10] http://solidity.readthedocs.io/en/develop/style-guide.html
[11] https://marketplace.visualstudio.com/items?itemName=JuanBlanco.solidity
[12] https://plugins.jetbrains.com/plugin/10177-solidity-solhint

- All'interno di un contratto le dichiarazioni di funzione vanno separate da una singola riga vuota;
- Le righe vuote possono essere omesse nel caso di gruppi di funzioni *one-liner*$_g$.

```
pragma solidity >=0.4.0 <0.6.0;

contract A {
  function spam() public pure;
  function ham() public pure;
}


contract B is A {
  function spam() public pure {
    // ...
  }

  function ham() public pure {
    // ...
  }
}
```

- **Lunghezza massima righe:** il massimo numero di caratteri consentito è 99, ciò aiuta a semplificare le attività di analisi e rilettura del codice;
- **Le righe racchiuse da parentesi devono essere conformi alle seguenti linee guida**:
  - Il primo argomento non deve essere collegato alla parentesi di apertura;
  - Deve essere usato un solo rientro per l'indentazione;
  - Ogni argomento dovrebbe possedere una propria riga.
  - L'elemento terminante con `);` dovrebbe essere posizionato per ultimo in una riga propria.

```
thisFunctionCallIsReallyLong(
  longArgument1,
  longArgument2,
  longArgument3
);
```

- **Ordinamento delle funzioni:** le funzioni vanno raggruppate secondo la loro visibilità e ordine:
  - `constructor`

- external
- public
- internal
- private

```
pragma solidity >=0.4.0 <0.6.0;

contract A {
  constructor() public {
    // ...
  }

  function() external {
    // ...
  }

  // External functions
  // ...

  // External functions that are view
  // ...

  // External functions that are pure
  // ...

  // Public functions
  // ...

  // Internal functions
  // ...

  // Private functions
  // ...
}
```

- **Spaziature in espressioni:** sono da evitare le spaziature superflue all'interno di espressioni, di seguito una serie di esempi.

```
// OK
spam(ham[1], Coin({name: "ham"}));

// NO
spam( ham[ 1 ], Coin( { name: "ham" } ) );

// OK
function spam(uint i, Coin coin) public;

```

```
10      // NO
11      function spam(uint i , Coin coin) public ;
```

- **Strutture di controllo:** Le parentesi che indicano il corpo di un contratto, libreria, funzioni e strutture dovrebbero:
  - Aprirsi sulla stessa riga della dichiarazione;
  - Chiudersi sulla propria riga allo stesso livello di indentazione dell'inizio della dichiarazione;
  - La parentesi di apertura, inoltre, dovrebbe essere preceduta da una singola spaziatura.

```
1       // OK
2       pragma solidity >=0.4.0 <0.6.0;
3
4       contract Coin {
5         struct Bank {
6           address owner;
7           uint balance;
8         }
9       }
10
11      // NO
12      pragma solidity >=0.4.0 <0.6.0;
13
14      contract Coin
15      {
16          struct Bank {
17                address owner;
18                uint balance;
19          }
20      }
21
22      // OK
23      if (...) {
24      ...
25      }
26
27      for (...) {
28      ...
29      }
30
31      // NO
32      if (...)
33      {
34      ...
35      }
36
37      while(...){
38      }
```

```
39
40      for (...) {
41      ...;}
```

- **Stringhe**: utilizzare sempre i doppi apici per le stringhe;

```
1   // OK
2   str = "foo";
3   str = "Hamlet says, 'To be or not to be...'";
4
5   // NO
6   str = 'bar';
7   str = '"Be yourself; everyone else is already taken." -Oscar ↩
      Wilde';
```

- **Ordine di Layout**: gli elementi che costituiscono un contratto scritto in *Solidity*$_g$ devono essere posizionati secondo questo ordine di layout:
  - Pragma statements
  - Import statements
  - Interfaces
  - Libraries
  - Contracts

  All'interno di ogni contratto, libreria o interfaccia va seguito il seguente ordine:
  - Type declarations
  - State variables
  - Events
  - Functions

2.2.4.4 Convenzione di codifica React/JSX

Le convenzioni di stile da applicare alla sintassi *JSX*$_g$ si basano principalmente sugli standard JavaScript. Nonostante la sintassi risulti simile, JSX presenta alcune particolarità che meritano essere normate.

- **Regole di base**:
  - Va inclusa una sola componente React per ciascun file;
  - Non utilizzare il metodo `React.createElement`. L'uso è consentito solo in fase di inizializzazione dell'app da un file non JSX.

- **Class vs React.createClass**: la sintassi accettata per l'istanziazione di classi è: `class extends React.Component` e non `React.createClass`

```
// bad
const Listing = React.createClass({
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
});

// good
class Listing extends React.Component {
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
}
```

- **Nomi**:
  - Per i nomi di file va utilizzato il *PascalCase*$_g$. E.g. ReservationCard.jsx;
  - Va utilizzato il *PascalCase*$_g$ per le componenti React e il camelCase per le loro istanze.

```
// bad
import reservationCard from './ReservationCard';

// good
import ReservationCard from './ReservationCard';

// bad
const ReservationItem = <ReservationCard />;

// good
const reservationItem = <ReservationCard />;
```

- **Dichiarazioni**: non va utilizzata la keyword `displayName` per assegnare nomi alle componenti. Va invece assegnato un nome attraverso riferimento.

```
// bad
export default React.createClass({
  displayName: 'ReservationCard',
  // stuff goes here
});

// good
export default class ReservationCard extends React.Component {
```

```
9       }
```

- **Spaziature**: va sempre aggiunta una spaziatura singola nella chiusura di un self-closing tag.

```
1       // bad
2       <Foo/>
3
4       // very bad
5       <Foo                    />
6
7       // bad
8       <Foo
9       />
10
11      // good
12      <Foo />
```

- **Parentesi**: i tag JSX che occupano più di una riga vanno racchiusi tra parentesi.

```
1       // bad
2       render() {
3         return <MyComponent variant="long body" foo="bar">
4         <MyChild />
5         </MyComponent>;
6       }
7
8       // good
9       render() {
10        return (
11        <MyComponent variant="long body" foo="bar">
12        <MyChild />
13        </MyComponent>
14        );
15      }
16
17      // good, when single line
18      render() {
19        const body = <div>hello</div>;
20        return <MyComponent>{body}</MyComponent>;
21      }
```

- **Tag**: i tag che non possiedono figli devono sempre essere chiusi con un self-closing tag.

```
1     // bad
2     <Foo variant="stuff"></Foo>
3
```

```
4   // good
5   <Foo variant="stuff" />
```

- **Ordinamento**:
  - Ordinamento per `class extends React.Component`:
    1. metodi `static`;
    2. `constructor`;
    3. `getChildContext`
    4. `componentWillMount`;
    5. `componentDidMount`;
    6. `componentWillReceiveProps`;
    7. `shouldComponentUpdate`;
    8. `componentWillUpdate`;
    9. `componentDidUpdate`;
    10. `componentWillUnmount`;
    11. clickHandlers o eventHandlers come `onClickSubmit()` o `onChangeDescription()`;
    12. metodi `getter` per `render` come `getSelectReason()` o `getFooterContent()`;
    13. metodi render opzionali come `renderNavigation()` o `renderProfilePicture()`;
    14. `render`.

2.2.5 Metriche

In questa sezione verranno esposte le metriche relative allo sviluppo dell'applicazione Soldino. La classificazione di queste metriche viene esposta nella sezione 3.2.3 del documento corrente.

2.2.5.1 [MS01] Complessità ciclomatica per metodo

Viene utilizzata per misurare la complessità dei moduli, metodi e classi del progetto software. Essa viene calcolata utilizzando un grafo per il controllo del flusso: i nodi rappresentano i blocchi di codice che non possono essere separati, mentre gli archi che connettono due nodi indicano se il secondo blocco di istruzioni può essere eseguito subito dopo il primo.
La metrica si basa su un indice numerico, dove valori troppo alti indicano un'elevata complessità del codice con probabile scarsa manutenibilità. Valori troppo bassi, invece, potrebbero indicare scarsa efficienza.

$$v(G) = e - n + 2p$$

dove

- *v(G)* è la complessità del grafo G;
- *e* è il numero di archi del grafo;
- *n* il numero di nodi del grafo;
- *p* il numero di componenti connesse.

2.2.5.2 [MS02] Livelli di annidamento

Sono i livelli di statement e cicli annidati all'interno di uno stesso blocco di codice. Un valore alto indica una complessità elevata e quindi una più difficile manutenibilità.

2.2.5.3 [MS03] Attributi per classe

Un numero elevato di attributi presenti nella classe potrebbe rappresentare una errata progettazione della stessa, e quindi far emergere la necessità di una maggiore circoscrizione delle classi.

2.2.5.4 [MS04] Numero di parametri per metodo

Un elevato numero di parametri potrebbe rappresentare una errata progettazione del metodo.

2.2.5.5 [MS05] Linee di codice per linee di commento

Rappresenta il rapporto tra linee di codice e linee commentate, utile per determinare la manutenibilità e la leggibilità del codice sorgente.

2.2.5.6 [MS06] Code Coverage

La valutazione di questa metrica è una valutazione complessiva di altre quattro metriche:

- **Function coverage**: verifica che ogni funzione sia stata chiamata;
- **Statement coverage**: verifica che ogni statement del codice sia stato eseguito;
- **Branch coverage**: verifica che tutti i possibili branch (derivanti da if e case statement) siano stati eseguiti;
- **Lines coverage**: verifica che ogni linea sia stata eseguita e/o percorsa.

Queste metriche vengono misurate in maniera automatica con gli strumenti descritti in §3.5.7.3.

2.2.5.7 [MS07] Copertura dei requisiti obbligatori

Indica la percentuale requisiti obbligatori coperti dall'implementazione: questa metrica è utile per capire a che punto dello sviluppo ci si trova. La sua formula di misurazione è:

$$ORC = \frac{NORC}{TOR} * 100$$

dove NORC è il numero di requisiti obbligatori soddisfatti e TOR è il numero di requisiti obbligatori totali.

2.2.5.8 [MS08] Copertura dei requisiti desiderabili

Indica la percentuale requisiti desiderabili e opzionali coperti dall'implementazione. La sua formula di misurazione è:

$$DRC = \frac{NDRC}{TDR} * 100$$

dove NDRC è il numero di requisiti desiderabili e opzionali soddisfatti e TDR è la somma del numero di requisiti desiderabili e opzionali totali.

2.2.5.9 [MS09] Errori gestiti correttamente

Indica la percentuale di funzionalità in grado di gestire correttamente gli errori che potrebbero verificarsi durante l'esecuzione dell'applicazione. La sua formula di misurazione è:

$$CMF = \frac{DF}{FTN} * 100$$

dove DF è il numero di errori evitati durante i test eseguiti e FTN è il numero di test che prevedono l'esecuzione di operazioni che causerebbero un errore.

2.2.5.10 [MS10] Tempo di risposta

Indica il tempo medio che intercorre fra la richiesta di una funzionalità da parte di un utente e la restituzione del risultato. La sua formula di misurazione è:

$$RT = \frac{\sum_{i=1}^{n} T_i}{n}$$

dove T_i è il tempo che intercorre fra la richiesta i di una funzionalità e il ritorno del risultato.

2.2.5.11 [MS11] Percentuale di failure

Indica la percentuale di testing dell'applicazione che si sono conclusi con un failure. La sua formula di misurazione è:

$$FP = \frac{NF}{NT} * 100$$

dove NF è il numero di failure rilevati durante l'attività di testing e NT è il numero di testing dell'applicazione eseguiti.

2.2.5.12 [MS12] Browser supportati

Indica i browser pienamente supportati dall'applicazione, con l'inclusione della versione, ad esempio:

- *Firefox v9.1*;
- *Chrome v61.*

2.2.6 Strumenti

Per la scrittura e l'esecuzione del codice il gruppo ha deciso di utilizzare i seguenti strumenti, alcuni dei quali suggeriti direttamente dalla Proponente.

- **Node.js**: https://nodejs.org/it/
  Node.js è una runtime di JavaScript Open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript Server-side, costruita sul motore JavaScript V8 di Google Chrome.

  NodeJs ha un'architettura orientata agli eventi che rende possibile l'I/O asincrono. Questo design punta ad ottimizzare il Throughput e la scalabilità nelle applicazioni web con molte operazioni di input/output, è inoltre ottimo per applicazioni web Real-time.

- **Truffle**: https://truffleframework.com/
  Truffle è un ambiente di sviluppo e framework per il test di applicazioni che utilizzano la Blockchain.

  Truffle offre un set di strumenti per la gestione semplificata del ciclo di vita degli Smart Contracts: dal testing automatico di Smart Contracts sviluppati in Solidity fino al loro deployment.

- **Surge**: https://surge.sh/
  Strumento usato per il deployment dell'applicazione su un server web.

3 Processi di supporto

3.1 Documentazione

3.1.1 Descrizione

In questo capitolo verranno descritte tutte le norme stabilite dal gruppo e le decisioni prese riguardo la scrittura, la verifica e l'approvazione di tutti i documenti ufficiali. Tali norme sono state redatte al fine di standardizzare la stesura dei documenti da parte del team.

3.1.2 Ciclo di vita documentazione

Ogni documento ufficiale attraversa tre stadi: Sviluppo, Verifica ed Approvazione.

- **Sviluppo**: questo stadio parte dalla stesura dello scheletro del documento, con l'obiettivo di agevolare i redattori nell'elaborazione dei contenuti, fornendo loro una base comune su cui lavorare. Ogni redattore elabora parti distinte in maniera autonoma e indipendente, assegnate individualmente tramite $ticket_{g}$, le quali verranno poi incorporate nel documento finale. Il passaggio alla fase di $verifica_{g}$ viene effettuato tramite notifica da parte dei redattori, nel momento in cui avranno completato le loro parti, al Responsabile di Progetto;
- **Verifica**: in questo stadio i verificatori incaricati dal Responsabile di Progetto controllano la correttezza, sia sintattica che grammaticale, del documento completo. Inoltre hanno il compito di assicurarsi che le singole parti del documento, sviluppate nel primo stadio, siano assemblate in maniera logica e accurata. Qualora i verificatori convalidino il documento questo entrerà nella fase di approvazione, dove verrà rilasciato ufficialmente; in caso contrario sarà compito del Responsabile di Progetto riassegnare il documento ai redattori, affinché che lo modifichino in vista di una successiva fase di verifica;
- **Approvazione**: questo processo è prerogativa del responsabile di progetto, il quale si accerterà che ogni documento rispetti gli standard definiti nelle Norme di Progetto. In caso di esito positivo, il documento potrà poi essere rilasciato ufficialmente.

3.1.3 Separazione documenti interni ed esterni

I documenti ufficiali sono raggruppati in base al loro destinatario. Ogni documento può essere rivolto ad un soggetto interno o esterno al gruppo; per tale motivo sono stati suddivisi in:

- **Interni**: tutti i documenti rivolti esclusivamente ai membri del gruppo, redatti in lingua italiana;
- **Esterni**: tutti i documenti rivolti ai proponenti o ai soggetti esterni al gruppo; ogni documento viene redatto in lingua italiana, ad eccezione del manuale utente che verrà redatto in lingua inglese, in modo da facilitarne la lettura da parte del proponente e degli utenti finali.

3.1.4 Nomenclatura documenti

Tutti i documenti ufficiali pubblicati seguiranno il sistema di Semantic Versioning 2.0.0[13], che, dato un documento con numero di versione MAJOR.MINOR.PATCH, incrementa la:

- **MAJOR**: ogniqualvolta si attuano modifiche radicali nei contenuti;
- **MINOR**: quando vengono apportate modifiche o incrementi a singole sezioni;
- **PATCH**: quando vengono corretti errori di natura ortografica o sintattica.

Ogni documento seguirà inoltre il seguente modello di nomenclatura:

- **NomeDocumento**: il nome del documento corrente, senza spazi e con le iniziali di ogni parola che lo compone in lettera maiuscola;
- **vMAJOR.MINOR.PATCH**: la versione del documento corrente, composta secondo le regole descritte sopra.

3.1.5 Documenti correnti

In questa sezione vengono elencati i documenti che costituiscono la documentazione ufficiale per il progetto:

- **Analisi dei Requisiti** (sigla AR, utilizzo esterno): documento che riporta le attività preliminari allo sviluppo del *capitolato*$_{\text{g}}$ Soldino, il cui scopo è quello di definire le funzionalità che il prodotto deve offrire, ovvero i requisiti che devono essere soddisfatti dal software sviluppato;
- **Glossario** (sigla Gl, utilizzo esterno): raccolta dei termini inerenti al progetto che possono essere causa di ambiguità o di difficile comprensione;
- **Norme di Progetto** (sigla NdP, utilizzo interno): documento in cui vengono descritti gli standard e le norme che i membri di *TWB*$_{\text{g}}$ dovranno adottare durante lo svolgimento delle attività progettuali;
- **Piano di Progetto** (sigla PdP, utilizzo interno): documento in cui vengono analizzate e descritte le scelte intraprese nella gestione delle risorse temporali e umane;
- **Piano di Qualifica** (sigla PdQ, utilizzo esterno): documento che descrive gli standard qualitativi che il prodotto finale dovrà raggiungere;
- **Studio di Fattibilità** (sigla SdF, utilizzo interno): documento che descrive le motivazioni che hanno portato il gruppo TWB alla scelta del capitolato Soldino.

3.1.6 Norme

In questo capitolo verranno descritte le norme adottate nella struttura dei documenti ufficiali e nello stile tipografico. Per facilitare la stesura dei documenti ufficiali, ad eccezione dei verbali e della lettera di presentazione, è stato creato un template *LaTeX*$_{\text{g}}$.

[13] url: https://semver.org/

3.1.6.1 Struttura dei documenti

- **Frontespizio**: la prima pagina di ogni documento, che contiene:
  - Logo del gruppo;
  - Titolo del documento;
  - Nome del gruppo;
  - Data dell'ultima modifica;
  - Informazioni sul documento (versione attuale, collaboratori, uso interno o esterno, a chi è rivolto);
  - Breve descrizione del documento.
- **Diario delle modifiche**: la seconda pagina di ogni documento, in cui vengono riportate tutte le modifiche effettuate allo stesso, dalla prima stesura alla verifica ed approvazione;
- **Indice**: in queste pagine sono indicate tutte le sezioni e sottosezioni presenti nel documento, indicando per ognuna il numero di capitolo, il titolo e il numero di pagina;
- **Introduzione**: una breve introduzione contenente lo scopo del documento, un riferimento al Glossario e i riferimenti normativi ed informativi;
- **Contenuto**: il contenuto del documento.

3.1.6.2 Norme tipografiche

- **Intestazione**: ogni pagina del documento, ad eccezione del frontespizio, contiene un'intestazione con il logo del gruppo a sinistra e la casella e-mail del gruppo a destra;
- **Piè di pagina**: in ogni pagina del documento, ad eccezione del frontespizio, sono presenti a piè di pagina il titolo del documento e il nome del gruppo a sinistra, e il numero di pagina attuale su pagine totali a destra;
- **Formattazione**:
  - Corsivo: per dare enfasi ad una o più parole, indicare i nomi dei documenti e dei termini presenti nel Glossario, e per mostrare prestiti linguistici che non sono presenti sui dizionari di italiano;
  - Grassetto: per indicare le voci di un elenco puntato che sono seguite da una descrizione, così come il titolo dei capitoli e delle sezioni;
  - Azzurro: per evidenziare i link esterni.
- **Virgolette**: gli apici sono utilizzati per caratteri singoli (e.g. 'c'), mentre le doppie virgolette sono utilizzate per racchiudere una o più parole (e.g. "parola") con significato particolare o figurato, ma anche per delimitare citazioni;

- **Parentesi**: le parentesi tonde sono utilizzate per dare precisazioni ed esempi, le parentesi quadre sono utilizzate per indicare uno standard (e.g. [ISO/IEC 12207]) o una sigla (e.g. [UC2.3]) , mentre le graffe per mostrare un commento lasciato da un membro del gruppo nel documento;
- **Elenchi**: un elenco può contenere delle voci in grassetto, seguite dal carattere ':' (non in grassetto) e la loro relativa descrizione, oppure delle frasi; in entrambi i casi, ogni elemento dell'elenco deve avere il primo carattere in maiuscolo ed essere seguito dal carattere ';', ad eccezione dell'ultimo, che sarà seguito da '.'; questo vale anche per i sottoelenchi, che dovranno comunque terminare con un '.';
- **Note a piè di pagina**: ogni nota a piè di pagina deve avere il primo carattere in maiuscolo, ad eccezione di link esterni, ed essere numerata seguendo l'ordine di apparizione all'interno della pagina;
- **Slash**: lo slash viene utilizzato per indicare due termini interscambiabili all'interno della frase;
- **Formati**:
  - **Date**: ogni data presente nei documenti deve essere scritta nel formato AAAA-MM-GG, dove AAAA indica l'anno, MM indica il mese e GG indica il giorno;
  - **Tabelle**: tutte le tabelle presenti nei documenti devono avere nella prima riga dei termini che definiscano le colonne che le compongono; se una tabella si estende per più di una pagina, la riga con le sue componenti deve essere riportata in ogni pagina in cui si estende.
- **Riferimenti informativi**: tutti i riferimenti informativi presenti nei documenti devono essere riportati in una nota a piè di pagina.

Di seguito è riportata una lista degli errori più comuni commessi nella stesura della documentazione, in modo da evidenziarli al fine di evitarli in futuro:

- **Punto e virgola alla fine di un elenco**: dove andrebbe invece un punto;
- **Punteggiatura in grassetto**: solamente le parole possono avere questo tipo di formattazione;
- **Lettere successive alla prima nei titoli in maiuscolo**: solamente i nomi propri devono cominciare con un carattere in maiuscolo;
- **Standard e sigle non racchiuse tra parentesi quadre**: questi elementi devono sempre essere delimitati da delle parentesi quadre.

3.1.7 Documentazione del codice sorgente

Il codice sorgente deve essere documentato per evitare ambiguità e rendere efficace l'attività di *verifica*$_g$.

3.1.7.1 Documentazione del codice sorgente JavaScript e React/JSX

Ogni file con estensione *.js* deve essere documentato attraverso lo strumento *JSDoc*$_g$[14] in versione 3.

JSDoc$_g$ è un *linguaggio di markup*$_g$ usato per aggiungere annotazioni a file sorgenti *JavaScript*$_g$. I file contenenti annotazioni *JSDoc*$_g$ possono essere processati da appositi tool che producono documentazione accessibile in formato *HTML*$_g$ o *RTF*$_g$.

JSDoc$_g$ prevede una sintassi specifica; viene qui riportato un estratto a titolo esemplificativo:

```
1  /**
2  * Creates an instance of Circle.
3  *
4  * @constructor
5  * @author: moi
6  * @this {Circle}
7  * @param {number} r The desired radius of the circle.
8  */
9  function Circle(r) {
10 /** @private */ this.radius = r;
11 /** @private */ this.circumference = 2 * Math.PI * r;
12 }
13
14 /**
15 * Creates a new Circle from a diameter.
16 *
17 * @param {number} d The desired diameter of the circle.
18 * @return {Circle} The new Circle object.
19 */
20 Circle.fromDiameter = function (d) {
21   return new Circle(d / 2);
22 };
23
24 /**
25 * Calculates the circumference of the Circle.
26 *
27 * @deprecated
28 * @this {Circle}
29 * @return {number} The circumference of the circle.
30 */
31 Circle.prototype.calculateCircumference = function () {
32   return 2 * Math.PI * this.radius;
33 };
```

[14] http://usejsdoc.org/

```
34
35    /**
36    * Returns the pre-computed circumference of the Circle.
37    *
38    * @this {Circle}
39    * @return {number} The circumference of the circle.
40    */
41    Circle.prototype.getCircumference = function () {
42      return this.circumference;
43    };
44
45    /**
46    * Find a String representation of the Circle.
47    *
48    * @override
49    * @this {Circle}
50    * @return {string} Human-readable representation of this Circle.
51    */
52    Circle.prototype.toString = function () {
53      return "A Circle object with radius of " + this.radius + ".";
54    };
```

3.1.7.2 Documentazione del codice sorgente Solidity

Il codice sorgente in *Solidity*$_g$ dovrà essere documentato attraverso commenti usando la notazione *Doxygen*$_g$[15] dove una o più righe iniziano con `///` e un commento multilinea inizia con `/**` e termina con `*/`.

Ad esempio, un semplice *Smart Contract*$_g$ arricchito con commenti è simile a quello riportato di seguito:

```
1     pragma solidity >=0.4.0 <0.6.0;
2
3     /// @author The Solidity Team
4     /// @title A simple storage example
5     contract SimpleStorage {
6     uint storedData;
7
8     /// Store `x`.
9     /// @param x the new value to store
10    /// @dev stores the number in the state variable `storedData`
11    function set(uint x) public {
12    storedData = x;
13    }
14
15    /// Return the stored value.
16    /// @dev retrieves the value of the state variable `storedData`
17    /// @return the stored value
18    function get() public view returns (uint) {
```

[15] http://www.doxygen.nl/

```
19    return storedData;
20    }
21    }
```

3.1.8 Struttura documentazione

Tutti i documenti sono redatti sulla base di un template sviluppato nel linguaggio *LaTeX*$_g$, in modo da facilitare il più possibile la loro stesura e renderli ordinati e leggibili.

3.1.9 Tracciamento casi d'uso, requisiti e test

Per automatizzare il tracciamento dei casi d'uso, dei relativi requisiti e dei test il gruppo si è avvalso di strumenti quali *Python*$_g$, *PHP*$_g$, MySQL[16] e DigitalOcean[17] per creare uno strumento di gestione del tracciamento di questi elementi.

3.1.10 Gestione termini Glossario

Il Glossario è un documento esterno che racchiude tutti i termini potenzialmente fraintendibili o di difficile comprensione, con le loro relative descrizioni, presenti in ogni documento ufficiale. I termini presenti nel Glossario devono essere contrassegnati, all'interno di tutti gli altri documenti, con una '$_g$' al pedice e la formattazione in corsivo (e.g. *LaTeX*$_g$). Per l'inserimento di nuovi termini all'interno del Glossario sono stati creati due comandi LaTeX:

- **Lettera**: aggiunge una nuova pagina al documento, in cui verranno elencati tutti i termini del Glossario che iniziano con quella lettera;
- **Parola**: aggiunge una nuova parola alla pagina del Glossario corrispondente alla prima lettera della parola.

La creazione di questi comandi è finalizzata ad un'agevolazione nella scrittura dei documenti e nell'inserimento di nuovi termini nel Glossario.
Nell'eventualità in cui nel Glossario siano presenti più voci indicanti lo stesso termine (e.g. "*UI*$_g$" e "*User Interface*$_g$"), la definizione deve essere riportata solo nella voce in cui il termine compare in maniera *estesa*, mentre tutte le altre voci devono essere dei riferimenti a questa definizione (e.g. per "*UI*$_g$" e "*User Interface*$_g$" la definizione deve essere riportata solo nella voce "*User Interface*$_g$", mentre la voce "*UI*$_g$" deve essere un riferimento alla voce "*User Interface*$_g$").

3.1.11 Metriche

In questa sezione verranno esposte le metriche relative alla documentazione dell'applicazione Soldino. La classificazione di queste metriche viene esposta nella sezione 3.2.3 del documento corrente.

[16] https://www.mysql.com/
[17] https://www.digitalocean.com/

3.1.11.1 [MD01] Rispetto delle norme

È compito dei Verificatori rileggere il documento prima dell'approvazione finale dello stesso. Verrà considerata come unità di misura dell'errore il numero di refusi presenti nel documento. È necessario seguire tutte le norme descritte nella sezione 3.1.6.2 del documento *NormeDiProgetto_ v3.0.0*, le quali riguardano:

- Intestazione;
- Piè di pagina;
- Formattazione;
- Virgolette;
- Parentesi;
- Elenchi;
- Note a piè di pagina;
- Slash;
- Formati di data;
- Riferimenti informativi.

Poiché queste norme sono decise internamente al gruppo di lavoro non è possibile automatizzare il processo, pertanto i Verificatori sono chiamati a effettuare verifiche manualmente, così da assicurarsi di correggere tutti gli eventuali refusi.

3.1.11.2 [MD02] Indice di Gulpease

Come metrica per la chiarezza è stato scelto di utilizzare l'*indice di Gulpease*$_{\text{g}}$, che è un numero oggettivo e misurabile.
Per la lingua italiana viene calcolato secondo la seguente formula:

$$89 + \frac{300 \times (NumeroDelleFrasi) - 10 \times (NumeroDelleLettere)}{NumeroDelleParole} \quad (1)$$

Dove:

- **NumeroDelleFrasi**: è il conteggio delle frasi separate da punto fermo all'interno del documento;
- **NumeroDelleLettere**: è il conteggio dei caratteri presenti nel documento esclusi gli spazi;
- **NumeroDelleParole**: è il conteggio delle parole presenti nel documento.

Il risultato ottenuto sarà un numero compreso tra 0 e 100, dove 0 indica un testo altamente incomprensibile mentre 100 un documento altamente comprensibile.
L'*indice di Gulpease*$_{\text{g}}$ si divide in tre scaglioni:

- **Inferiore a 80**: di difficile comprensione per chi ha la licenza elementare;

- **Inferiore a 60**: di difficile comprensione per chi ha la licenza media;
- **Inferiore a 40**: di difficile comprensione per chi ha un diploma superiore.

Per il calcolo di questo indice è stato sviluppato uno script in *Python*, in modo da generare automaticamente un report contenente gli esiti del calcolo per ciascun documento trattato.

3.1.11.3 [MD03] Correttezza del contenuto

Perché il documento sia di qualità è necessario che sia veritiero e non ambiguo, pertanto è richiesto che sia corretto anche nel contenuto.
Come metrica viene considerato il numero di errori contenutistici inseriti.

Questa metrica non è automatizzabile perciò è compito dei Verificatori la lettura del documento per almeno 2 volte così da eliminare tutti gli errori.

3.1.11.4 [MD04] Formula di Flesch

Questa metrica serve a misurare la leggibilità di un testo in inglese. Il modo di calcolare questo indice è il seguente:

$$F = 206,835 - (0,846 * S) - (1,015 * P)$$

dove:

- F è la leggibilità misurata secondo questi parametri;
- S è il numero delle sillabe, calcolato su un campione di 100 parole;
- P è il numero medio di parole per frase.

Più questo indice è alto e più il testo risulta semplice da leggere.

3.1.12 Ambiente

La scrittura dei documenti deve essere effettutata utilizzando il linguaggio *LaTeX*$_{\text{g}}$ negli ambienti *TexMaker*$_{\text{g}}$ e/o *TexStudio*$_{\text{g}}$.

3.2 Qualità

3.2.1 Descrizione

In questa sezione vengono descritte le classificazioni e le procedure adottate per il calcolo delle metriche illustrate nel *PianoDiQualifica_ v3.0.0*.

3.2.2 Classificazione dei processi

I processi istanziati nel progetto sono codificati con la seguente notazione **PROC[id][int]**, dove:

- **PROC**: è l'abbreviazione di processo;

- **id**: rappresenta la categoria di appartenenza del processo secondo lo standard [ISO/IEC 12207:2017];
  L'identificativo può assumere uno dei seguenti valori:
  - **A**: Agreement Processes;
  - **O**: Organizational project-Enabling Processes;
  - **M**: Technical Management Processes;
  - **T**: Technical Processes.
- **int**: rappresenta un numero incrementale a due cifre a partire da 01.

3.2.3 Classificazione delle metriche

Per garantire la qualità di processo e prodotto sono state adottate una serie di metriche, che devono rispettare la notazione **M[id][int]**, dove:

- **M**: è l'abbreviazione di metrica;
- **id**: è il codice identificativo associato alla tipologia di metrica e può assumere i seguenti valori:
  - **P**: indica una metrica di processo;
  - **D**: indica una metrica di documento;
  - **S**: indica una metrica del prodotto software;
  - **T**: indica una metrica di test.
- **int**: rappresenta un numero incrementale a due cifre a partire da 01.

3.2.4 Procedure

Per poter applicare il $PDCA_{\text{g}}$ correttamente è necessario innanzitutto che:

- I processi siano pianificati in modo accurato;
- Le risorse siano allocate in modo adeguato nei processi;
- I processi possano essere mantenuti sotto controllo.

Soddisfatte tali premesse vengono intrapresi i seguenti passi per ottimizzare i processi:

1. Individuazione degli obiettivi;
2. Descrizione e $analisi_{\text{g}}$ della situazione corrente;
3. Motivazione della risoluzione di miglioramento;
4. Raccolta dati tramite osservazione e $analisi_{\text{g}}$ del problema;
5. Verifica dell'attendibilità, oggettività e validità dei dati raccolti;
6. Campionamento dei dati in modo rappresentativo e consistente.

Al termine di queste azioni, viene formulato un piano per centrare l'obiettivo concordato. Il piano non solo identificherà ciò che bisogna fare, in quale sequenza andrà fatto e quando, ma anche le risorse da allocare alla singola attività.

Strumenti adatti a questo scopo sono *brainstorming*$_g$, *diagrammi di flusso*$_g$ e *diagrammi causa effetto*$_g$ che possono far emergere nuove idee di miglioramento e concorrono all'individuazione di possibili cause di inefficienze nei processi.

Successivamente, viene applicata la strategia correttiva ed implementato il piano di miglioramento.

In questa fase si sperimenta su bassa scala se la strategia può essere applicabile, e in caso positivo la si può estendere ad altri processi.

Conclusa l'implementazione delle modifiche pianificate, dopo un tempo sufficientemente lungo da permettere una raccolta dei dati ragionevole, si confronteranno i risultati ottenuti con quelli attesi e si rifletterà sugli esiti raggiunti così da migliorare l'approccio in base all'esperienza acquisita.

Raccolte sufficienti informazioni viene discusso se:

- Abbandonare i cambiamenti apportati poiché inefficaci o controproducenti;
- Adottare il cambiamento apportato e avviarlo su larga scala.

Infine, standardizzare il perfezionamento adottato in modo da rendere consolidate e irreversibili le azioni correttive.

3.3 Gestione delle attività di verifica

3.3.1 Pianificazione

La pianificazione temporale necessita di uno sguardo a ritroso a partire dagli obiettivi prefissati per completare in tempo adeguato il lavoro previsto. Per affrontare al meglio le scadenze il team si propone di completare i *deliverable*$_g$ richiesti per la revisione con un anticipo di 4 giorni al fine di applicare le relative procedure di controllo *qualità*$_g$.

- **Metrica**: si è deciso di utilizzare la Schedule Variance;
- **Soglia di accettabilità**: si è deciso di ritenere accettabile un ritardo massimo del 10% rispetto a quanto specificato nel documento *PianoDiProgetto_ v3.0.0*;
- **Soglia di ottimalità**: si ritiene un miglioramento rispetto all'obiettivo prefissato il caso in cui un lavoro venga portato a termine entro quanto specificato nel documento *PianoDiProgetto_ v3.0.0*.

Avendo l'obiettivo di rispettare le scadenze fissate nel documento *PianoDiProgetto_ v3.0.0*, è necessario che l'attività di *verifica*$_g$ della documentazione e del codice sia sistematica e ben organizzata. Applicando tali principi l'individuazione e la correzione degli eventuali errori avverrà il prima possibile, impedendo una rapida diffusione degli stessi. Ogni fase di redazione dei documenti e di *codifica*$_g$ deve essere preceduta da una fase di studio preliminare. Tale fase

ha lo scopo di ridurre la possibilità di commettere imprecisioni di natura concettuale e tecnica favorendo l'attività di *verifica*$_{\text{g}}$, dove saranno necessari minori interventi di correzione.

3.3.2 Responsabilità

3.3.2.1 Responsabile di Progetto Il Responsabile di Progetto rappresenta il team, per questo deve essere presente per tutta la durata dello sviluppo; per lo stesso motivo, è compito del Responsabile di Progetto interfacciarsi con i *Committenti*$_{\text{g}}$ e la Proponente.
Rientrano nelle sue responsabilità:

- La pianificazione;
- La gestione e coordinamento delle risorse umane;
- La creazione di ticket e l'assegnazione del relativo responsabile;
- Assicurarsi che vengano svolte le attività di *verifica*$_{\text{g}}$ seguendo il documento *NormeDiProgetto_ v3.0.0* senza che ci siano conflitti tra Redattori e Verificatori;
- L'approvazione e la distribuzione del documento.

3.3.2.2 Verificatore Il Verificatore è presente per tutta la durata del progetto, conosce le norme e ha capacità di giudizio e relazione.
Rientrano nelle sue responsabilità:

- Effettuare la *verifica*$_{\text{g}}$ dei documenti secondo quanto descritto dal Piano di Qualifica;
- Segnalare eventuali errori come descritto nel documento *NormeDiProgetto_ v3.0.0*;
- Sottoporre l'approvazione finale del documento al Responsabile di Progetto.

3.3.3 Strumenti ed azioni a supporto delle attività di verifica

Per la *verifica*$_{\text{g}}$ del prodotto si rivelano utili alcune risorse tecnologiche come:

- La documentazione di progetto in *LaTeX*$_{\text{g}}$ per la tracciabilità dei requisiti;
- La creazione di *diagrammi UML*$_{\text{g}}$;
- L'automatizzazione della *verifica*$_{\text{g}}$ dei documenti attraverso tool di controllo ortografico;
- La gestione dei test e le analisi sul codice;
- Il servizio *Travis CI*$_{\text{g}}$ di *continuous integration*$_{\text{g}}$;
- Il sistema di versioning *Git*$_{\text{g}}$;
- Il monitoraggio proattivo e passivo dello stato di progetto attraverso analisi continua dei processi.

Per una descrizione esaustiva della strumentazione utilizzata si veda il documento *NormeDiProgetto_ v3.0.0*.

3.4 Configurazione

3.4.1 Controllo di versione

3.4.1.1 Descrizione

Per le risorse versionabili è stato scelto come *Version Control System*$_g$ *Git*$_g$. Per quanto concerne lo scambio di documenti e materiale non versionabile è stata utilizzata l'applicazione *Discord*$_g$.

3.4.1.2 Struttura dei repository

In questa prima fase del progetto è stato configurato un *repository*$_g$ denominato "The-Walking-Bug", il cui utilizzo è rivolto alla gestione e al *versionamento*$_g$ dei primi documenti necessari alla RR. Per il versionamento del codice e delle risorse riguardanti lo sviluppo del prodotto verranno certamente creati altri *repository*$_g$ in un secondo momento.
La struttura delle directory presenti nel *repository*$_g$ attuale sono organizzate nel modo seguente:

- **Interni**:
  - NormeDiProgetto;
  - StudioDiFattibilità;
  - Verbali interni.
- **Esterni**:
  - AnalisiDeiRequisiti;
  - Glossario;
  - PianoDiProgetto;
  - PianoDiQualifica;
  - Verbali esterni.

All'interno di ciascuna cartella sopra elencata sono presenti ulteriori directory che organizzano le risorse per la scrittura dei file di documentazione *LaTeX*$_g$.

3.4.1.3 Ciclo di vita dei branch

Nel *sistema di versionamento*$_g$ vengono utilizzati dei meccanismi chiamati *branch*$_g$ per suddividere il flusso del lavoro svolto, in questo modo si ha una migliore gestione logica delle varie parti che devono essere sviluppate, inoltre permette di evitare che avvengano numerose modifiche in contemporanea sullo stesso *branch*$_g$ costringendo così lo sviluppatore a risolvere eventuali conflitti. L'utilizzo dei *branch*$_g$ avviene nel seguente modo:

- Deve essere creato un *branch*$_g$ per ogni documento che viene prodotto denominato NomeDocumento;
- Coloro che lavorano sul documento X devono eseguire i *commit* e i *push* sul *branch*$_g$ denominato X;

- Periodicamente, viene stabilita una data in cui viene eseguito il *merge*$_g$ dei *branch*$_g$ riguardanti i documenti sul master *branch*$_g$, al fine di riunificare il lavoro svolto e ripianificare le prossime aggiunte sui documenti;
- Ulteriori *branch*$_g$, utili per altre attività, possono essere creati previa discussione con tutti i membri del team.

3.4.1.4 Aggiornamento del repository

Questo paragrafo descrive il workflow del *repository*$_g$ attuale:

- Per ottenere le ultime modifiche presenti sul *repository*$_g$ in remoto si deve utilizzare il comando *git pull*. Nel caso in cui si verifichino dei conflitti si deve:
  - Utilizzare il comando *git stash* per accantonare momentaneamente le modifiche attuali;
  - Utilizzare il comando *git pull* per aggiornare il *repository*$_g$ in locale con i *commit* mancanti;
  - Utilizzare *git stash apply* per ripristinare le modifiche che si stavano apportando in precedenza.

  Seguendo questi tre passi si aggiornerà il *repository*$_g$ locale rispetto a quello remoto, mantenendo le modifiche che si stavano apportando;
- Utilizzare il comando *git add [files]* per aggiungere i file alla *staging area*$_g$;
- Utilizzare il comando *git commit* (eventualmente con il *flag*$_g$ -m per aggiungere una descrizione) per registrare le modifiche;
- Utilizzare il comando *git push* per inviare le modifiche al server che ospita il *repository*$_g$ remoto e renderle cosí disponibili agli altri collaboratori.

È necessario che, prima di eseguire il flow di comandi appena descritto, ci si assicuri che i file modificati siano stati compilati e abbiano prodotto il corrispondente file pdf, cosicché il *repository*$_g$ sia costantemente aggiornato. Inoltre questa attività permette di evitare l'introduzione di errori che impediscono la compilazione dei file.

3.4.1.5 Strumenti di supporto alla documentazione

Github$_g$ offre un sistema di gestione delle *release*$_g$, il quale si occupa di organizzare i prodotti creati nel progetto (e.g. documenti, file eseguibili, archivi). Viene quindi sfruttata questa funzionalità insieme alla *continuous integration*$_g$ mediante *Travis CI*$_g$; in questo modo si ha la possibilità di rendere disponibili le ultime versioni degli artefatti come specificato in seguito. L'utilizzo di questo strumento viene utilizzato principalmente nel momento in cui vengono apportate grosse modifiche al documento, quindi all'avvento di una MAJOR o di una MINOR oppure, se ritenuto strettamente necessario, ad una PATCH.

3.4.1.6 Rilascio dei documenti aggiornati

Questo paragrafo illustra i passaggi necessari per il rilascio della versione aggiornata dei documenti in formato pdf, utilizzando la funzionalità descritta nel paragrafo precedente. Di seguito la procedura:

- Seguire i passi indicati nella sezione 3.4.1.4;
- Utilizzare il comando *git tag vMAJOR.MINOR.PATCH* per creare il tag corrispondente al numero di *release*$_{\text{g}}$;
- Utilizzare il comando *git push origin –tagname* per pubblicare il nuovo tag sul *repository*$_{\text{g}}$ remoto;
- Utilizzare il comando *git push origin master* per pubblicare le modifiche apportate ai documenti e innescare il meccanismo di *Continuous Integration*$_{\text{g}}$ che permette di compilare automaticamente le risorse *LaTeX*$_{\text{g}}$ e rilasciare la nuova versione nell'area predisposta da *Github*$_{\text{g}}$.

3.4.1.7 Strumenti

- **Client Git**: ciascun membro del team potrà scegliere lo strumento che preferisce; quelli utilizzati attualmente sono *Github Desktop*$_{\text{g}}$, *GitKraken*$_{\text{g}}$, *Git bash*$_{\text{g}}$;
- **Server Git**: il *repository*$_{\text{g}}$ attuale e quelli futuri è *hostato*$_{\text{g}}$ su *Github*$_{\text{g}}$ perché offre una buona integrazione con strumenti esterni come *Travis CI*$_{\text{g}}$ e *Webhook*$_{\text{g}}$ da utilizzare con l'applicazione *Discord*$_{\text{g}}$, utili per notificare le attività che il team svolge. Inoltre, sfruttando la licenza studenti, è possibile rendere questi *repository*$_{\text{g}}$ privati.

3.5 Verifica

3.5.1 Descrizione

La verifica è uno dei processi fondamentali dello sviluppo di un prodotto poiché permette di accertare l'assenza di errori. In questa sezione verranno introdotti gli strumenti che saranno poi utilizzati per la *verifica*$_{\text{g}}$ del codice e dei documenti prodotti. Nell'attuale fase del progetto la *verifica*$_{\text{g}}$ richiesta è focalizzata sui documenti e sui diagrammi.

3.5.2 Analisi statica

L'analisi statica è una procedura che può essere applicata tanto al codice sorgente quanto alla documentazione, al fine di accertare che le attività svolte non abbiano introdotto degli errori.
Vi sono due modalità per eseguire la verifica:

- **Walkthrough**: questa tecnica assume che gli attori coinvolti in questa attività non mirino a trovare degli errori specifici, ma eseguono una lettura completa del documento in questione per rilevare errori di qualsiasi tipo. Viene sempre svolto, su ciascun documento prodotto, per la verifica dei contenuti dei documenti e per la loro correttezza morfosintattica;
- **Inspection**: questa tecnica invece prevede una lettura mirata, in cui gli attori hanno già un'idea della tipologia di errori che devono essere cercati e corretti, sulla base di uno storico che ne elenca i più comuni.

3.5.3 Analisi dinamica

L'analisi dinamica è una tecnica che permette di verificare la correttezza del codice con l'ausilio di test automatici, pertanto ne richiede la sua esecuzione. Per quanto riguarda i documenti quanto appena descritto non è applicabile.

3.5.4 Classificazione dei test

I test eseguiti durante il progetto verranno codificati con la seguente notazione **T[id][int]**, dove:

- **T**: è l'abbreviazione di test;
- **[id]**: è il codice identificativo della tipologia di test, e può assumere i valori:
  - **U**: indica un test di unità;
  - **I**: indica un test di integrazione;
  - **S**: indica un test di sistema;
  - **V**: indica un test di validazione;
  - **R**: indica un test di regressione.
- **[int]**: è il codice identificativo del test specifico, indicato con un numero incrementale a partire da 01.

Un test inoltre assume uno stato tra quelli seguenti:

- Non implementato;
- Implementato;
- Non superato;
- Superato.

Per l'esecuzione dei test sulle varie parti del prodotto sono stati scelti i seguenti strumenti:

- **React**: Jest, Enzyme;
- **Redux**: Jest;
- **Solidity**: Mocha, Chai.

3.5.5 Metriche

In questa sezione verranno esposte le metriche relative alla verifica dell'applicazione Soldino. La classificazione di queste metriche viene esposta nella sezione 3.2.3 del documento corrente.

3.5.5.1 [MT01] Percentuale di test passati Questa metrica è utile per capire a che punto dello sviluppo della componente ci si trova: un numero basso indica che si è ancora

agli inizi, un numero alto indica che la componente è quasi completamente sviluppata. La sua formula di misurazione è:

$$PPT = \frac{PT}{ET} * 100$$

dove PT indica il numero di test passati e ET il numero di test eseguiti.

3.5.5.2 [MT02] Percentuale di test falliti Indica la percentuale di test falliti: questa metrica è la complementare della precedente. La sua formula di misurazione è:

$$PFT = \frac{FT}{ET} * 100$$

dove FT indica il numero di test falliti e ET il numero di test eseguiti.

3.5.5.3 [MT03] Tempo medio di risoluzione degli errori Indica il tempo medio impiegato per risolvere un bug: questa metrica è utile per capire l'impatto dell'introduzione di un bug nei tempi di sviluppo del prodotto. La sua formula di misurazione è:

$$MER = \frac{TTBR}{TBN} * 100$$

dove TTBR indica il tempo totale impiegato per la risoluzione di bug e TBN il numero totale di bug trovati.

3.5.5.4 [MT04] Efficienza della progettazione dei test Indica il tempo di scrittura di un test: un numero troppo elevato per questa metrica potrebbe indicare che si stanno sviluppando test troppo specifici o troppo complessi. La sua formula di misurazione è:

$$TDEI = \frac{NTD}{TDT}$$

dove NTD indica il numero di test sviluppati e TDT il tempo in cui sono stati scritti.

3.5.5.5 [MT05] Efficacia della progettazione dei test Indica il rapporto fra i bug trovati durante l'esecuzione dei test e di quelli trovati durante l'utilizzo dell'applicazione. Un numero troppo basso per questa metrica potrebbe significare una scarsa attenzione nello sviluppo dei test. La sua formula di misurazione è:

$$TDEA = \frac{TBF}{TTBF} * 100$$

dove TBF indica il numero di bug trovati durante l'esecuzione dei test, mentre TTBF il numero totale di bug trovati durante l'esecuzione dei test e dell'applicazione.

3.5.5.6 [MT06] Percentuale requisiti coperti Indica la percentuale di requisiti coperti dai test rispetto al numero totale di requisiti: questa metrica è utile per capire quante parti del prodotto hanno un test specifico associato, e non dà indicazioni sullo stato di avanzamento dello sviluppo della soluzione per il requisito. La sua formula di misurazione è:

$$CRP = \frac{CR}{TR} * 100$$

dove CR indica il numero di requisiti coperti e TR il numero di requisiti totali.

3.5.6 Verifica diagrammi UML

I verificatori hanno il compito di controllare che i *diagrammi UML*$_g$ prodotti siano conformi allo standard adottato e siano corretti dal punto di vista semantico.

3.5.7 Strumenti

3.5.7.1 Documentazione

Per verificare la correttezza lessicale dei documenti i verificatori utilizzano i seguenti strumenti:

- Dizionario di *TeXStudio*$_g$;
- Dizionario di *TeXMaker*$_g$.

3.5.7.2 Analisi statica

- **ESLint**: https://eslint.org/;
- **Sass**: https://github.com/sasstools/sass-lint;
- **Solhint**: https://github.com/protofire/solhint.

3.5.7.3 Analisi dinamica

- **Jest**: https://airbnb.io/enzyme/docs/guides/jest.html;
- **Mocha**: https://mochajs.org/;
- **Chai**: https://www.chaijs.com/;
- **Solidity Coverage**: https://www.npmjs.com/package/solidity-coverage;

3.5.7.4 Continuos Integration

Travis CI: https://travis-ci.org/
Travis CI è stato scelto come strumento per applicare l'integrazione continua al progetto, sia per l'applicativo software che per la documentazione redatta in Latex.
Travis CI si occupa di eseguire la build del progetto prelevandone l'ultima versione dal sistema di Version Control *Git*$_g$. Un'apposito script di configurazione viene poi lanciato da Travis CI in modo da svolgere specifici comandi durante la fase di compilazione del progetto.

3.5.7.5 Continuous Inspection

SonarQube: https://www.sonarqube.org/
La pagina "Issues" del *repository* permette di visualizzare il dettaglio dei problemi trovati, di identificare dove si trovano, quando sono stati aggiunti e chi li ha introdotti.

3.6 Validazione

3.6.1 Descrizione

La validazione è un altro processo fondamentale, che si occupa di accertare che quanto prodotto sia conforme alle attese e alle specifiche.

3.6.2 Procedure

I processi per eseguire l'attività di validazione sono i seguenti:

- **Tracciamento**: attività automatizzata che tiene traccia dei test svolti corredato da $report_{\mathrm{g}}$ forniti dai verificatori;
- **Analisi risultati**: il responsabile consulta i report prodotti dai verificatori e in seguito prende queste decisioni:
  - Accettazione della validazione;
  - Chiedere nuovamente l'esecuzione dei test, anche completa, e realizzarne di nuovi.

4 Processi organizzativi

4.1 Processi di coordinamento

4.1.1 Comunicazioni

In questa sezione vengono specificate le norme a cui attenersi per una comunicazione omogenea all'interno e all'esterno del gruppo (e.g. comunicazione con i Committenti).

4.1.1.1 Comunicazioni interne

Per le comunicazioni interne al gruppo sono stati scelti tre canali principali:

- ***Telegram*$_{\mathbf{g}}$**: per accordarsi sugli incontri e scambiarsi eventuali comunicazioni tempestive;
- ***Discord*$_{\mathbf{g}}$**: strumento usato principalmente per conferenze vocali ufficiali con il gruppo;
- ***Trello*$_{\mathbf{g}}$**: strumento utilizzato per la gestione dei task, scadenze e *ticketing*$_{\mathrm{g}}$.

Sulla piattaforma *Discord*$_{\mathrm{g}}$ sono stati inoltre creati dei canali di comunicazione per agevolare la comunicazione dei sotto-gruppi che sono venuti a formarsi:

- **technology-baseline**: Canale dedicato alla formazione dei membri del gruppo sulle tecnologie utilizzate;
- **documentazione**: Canale dedicato all'organizzazione dei membri del gruppo per quanto riguarda la correzione e all'incremento dei documenti.

4.1.1.2 Comunicazioni esterne Per le comunicazioni ufficiali con la Proponente Red Babel, e i Committenti prof. Tullio Vardanega e prof. Riccardo Cardin, vengono utilizzati tre canali:

- *Gmail*$_{\mathrm{g}}$;
- *Slack*$_{\mathrm{g}}$;
- *Hangouts*$_{\mathrm{g}}$.

Per le comunicazioni via mail è stata creata una casella apposita consultabile da tutti i membri del gruppo: thewalkingbug.swe@gmail.com.
Per la scrittura delle mail è necessario seguire le seguenti regole:

- **Oggetto**: "[TWB - Soldino]" e una breve descrizione del cotenuto;
- **Apertura**: "Gentile [destinatario]," seguito da un a capo;
- **Corpo**: descrizione del contenuto della mail;
- **Chiusura**: "Distinti saluti," e a capo la firma dell'amministratore.

Per una comunicazione più immediata viene usato *Slack*$_{\mathrm{g}}$, dove i membri del gruppo TWB hanno accesso a un canale in cui sono presenti entrambi i membri del team di *Red Babel*$_{\mathrm{g}}$. Questo strumento è usato principalmente come chat testuale, quindi gli scopi di questo ambiente sono la risoluzione a domande semplici, l'accordarsi sulle riunioni e il notificare la

proponente sull'andamento del progetto.

Hangouts$_{\text{g}}$ è usato per effettuare chiamate di gruppo fra la Proponente e i membri di TWB. Le *call* su *Hangouts*$_{\text{g}}$ vengono intese come riunioni ufficiali.

4.1.2 Riunioni

Le riunioni sono di fondamentale importanza per la gestione del gruppo e degli obiettivi da conseguire.
Ogni riunione pianificata dal Responsabile viene formalizzata e calendarizzata all'interno del tool Google Calendar. Il Responsabile ha inoltre l'obbligo di inoltrare ai membri del team la richiesta di partecipazione a tale riunione. Ogni membro del team, alla ricezione della notifica di "invito a partecipare" alla riunione, è tenuto a confermare o negare la propria presenza tempestivamente.

4.1.2.1 Verbali di riunione

Ad ogni riunione deve essere redatto un verbale da un membro del gruppo che assume la funzione di Segretario.
Il verbale dovrà riassumere quanto discusso durante il meeting, sia esso fisico o telematico, e il documento creato sarà denominato come segue: TIPO_AAAA-MM-GG, dove "TIPO" può assumere i valori "INTERNO" o "ESTERNO".
Il documento dovrà avere la seguente struttura:

- **Informazioni incontro**:
  - Luogo;
  - Data;
  - Orario d'inizio;
  - Partecipanti.
- **Argomenti**: in questa sezione, mediante un elenco puntato, viene definito l'*ordine del giorno*$_{\text{g}}$;
- **Resoconto**: viene fatto il riassunto di quanto emerso durante la discussione dell'*ordine del giorno*$_{\text{g}}$ ed eventuali digressioni.

Nel documento saranno inoltre presenti e descritte le decisioni prese durante la riunione. Queste devono essere identificate con il seguente formato:

$$TVTipoNumeroVerbale.NumeroDecisione$$

dove *Tipo* può assumere i valori "I" per indicare un verbale interno e "E" per indicarne uno esterno.
È dovere del Responsabile infine riportare e tracciare queste decisioni come task sulla piattaforma *Trello*$_{\text{g}}$, in modo da rendere identificabili le motivazioni delle decisioni prese.

4.1.2.2 Riunioni interne

Data l'importanza per il gruppo di essere sempre aggiornato sull'andamento delle attività e vista la natura collaborativa del progetto, si ritiene necessario fare una riunione, anche non ufficiale, una volta a settimana.
Una riunione per essere considerata ufficiale deve attenersi alle seguenti regole:

- Il Responsabile di progetto deve accordarsi con il team per fissare una data e un orario per la riunione, con conseguente conferma da parte dei componenti;
- Presenza obbligatoria del Responsabile di progetto e di un Segretario;
- Partecipazione di almeno il 60% dell'organico;
- Deve essere redatto il verbale da parte del Segretario, attenendosi allo standard descritto in precedenza al paragrafo *4.1.2.1 Verbali di riunione*.

4.1.2.3 Riunioni esterne

Alle riunioni esterne deve essere obbligatoriamente presente il Responsabile di progetto e il Segretario.
Il Responsabile di progetto sarà il portavoce del team di sviluppo e con il Proponente discuterà di eventuali problematiche, obiettivi da raggiungere e/o eventuali modifiche da apportare al progetto.

4.1.2.4 Casi straordinari

Nell'eventualità in cui gli impegni lavorativi e universitari di ogni membro del gruppo costituiscano un ostacolo per il regolare svolgersi di riunioni fisiche, sia interne sia esterne, verranno considerate ufficiali anche quelle tenute con l'ausilio di strumenti telematici (e.g. *Discord*$_{\text{g}}$).

4.2 Processi di pianificazione

4.2.1 Ruoli di progetto

Per assicurare che tutti i membri del gruppo partecipino in egual misura e in modo collaborativo, ogni ruolo verrà assegnato a rotazione.
Il Responsabile di progetto in carica dovrà scegliere come e quando cambiare il ruolo di ogni persona al fine di non scaturire conflitti interni.
I ruoli sono i seguenti:

- Amministratore;
- Analista;
- Progettista;
- Programmatore;
- Responsabile di progetto;
- Verificatore.

4.2.1.1 Amministratore

L'Amministratore si occupa di fornire la strumentazione e un ambiente di lavoro adeguati per facilitare lo svolgimento delle attività da eseguire. I suoi doveri sono:

- Scrittura del documento *NormeDiProgetto*;
- Collaborazione alla realizzazione del documento *PianoDiProgetto*;
- Controllo del *versionamento*$_g$ e della configurazione dei prodotti;
- Controllo della *qualità*$_g$ sul prodotto.

4.2.1.2 Analista

Il ruolo dell'Analista è di vitale importanza in quanto deve capire a fondo il problema che viene sottoposto al team, così da stilare una lista di requisiti da realizzare per offrire una soluzione adeguata. I suoi compiti sono:

- Redigere il documento *StudioDiFattibilità*;
- Individuare tutti i requisiti richiesti, sia espliciti sia impliciti, e riportarli nel documento *AnalisiDeiRequisiti*;
- Svolgere un'*analisi SWOT*$_g$ sulle attività progettuali.

4.2.1.3 Progettista

Il Progettista si deve occupare del *design*$_g$ del progetto, scomponendolo in parti di semplice realizzazione, al fine di soddisfare i punti fissati nell'*Analisi dei Requisiti* in modo verificabile. Il suo ruolo è quello di progettare una soluzione facilmente *manutenibile*$_g$ seguendo delle *best practice*$_g$.

4.2.1.4 Programmatore

Il Programmatore deve realizzare il progetto come richiesto dal Progettista, dovrà quindi:

- Implementare il codice;
- Redigere la documentazione inerente al codice (*Manuale Manutentore*$_g$);
- Redigere la documentazione inerente al prodotto (*Manuale Utente*$_g$);
- Attenersi alle norme fissate per il *versionamento*$_g$ e la *manutenibilità*$_g$;
- Creare test adatti per verificare il corretto funzionamento del codice.

4.2.1.5 Responsabile di progetto

Il Responsabile di progetto è colui che detiene il potere decisionale. Ricadono su di lui le responsabilità di:

- Coordinamento delle attività;
- Fissare le scadenze;
- Approvazione finale della documentazione;

- Indire riunioni con conseguente convocazione dei membri del team;
- Creazione dell'OdG_g;
- Relazionarsi con la Proponente e i Committenti.

4.2.1.6 Verificatore

Il Verificatore deve svolgere un'attività continua di controllo sulla documentazione inerente al prodotto.
Deve inoltre assicurarsi che non vengano introdotti errori durante lo svolgimento delle attività per il raggiungimento degli obiettivi.

4.2.2 Ticketing

Per organizzare il lavoro di gruppo, il Responsabile di progetto usufruirà del servizio $Trello_g$ per assegnare i diversi compiti, i quali devono essere conformi agli obiettivi da raggiungere. Su $Trello_g$ è possibile creare dei task visibili in pubblicamente a chi ha accesso alla bacheca di lavoro, ed è quindi dovere del Responsabile crearla e aggiungerne i membri.

4.2.2.1 Creazione task

Per la creazione di un task bisogna seguire la seguente procedura:

1. Aggiungere una scheda nella $board_g$ "To Do";
2. Assegnare un titolo alla scheda;
3. Assegnare una descrizione al task;
4. Selezionare i membri incaricati alla risoluzione del task;
5. Stabilire una scadenza per il task.

È evenutalmente possibile aggiungere una checklist le cui voci dovranno essere spuntate dai singoli membri ogniqualvolta portino a termine un task assegnato loro.

4.2.2.2 Stato task

Un task può trovarsi in uno di questi stati:

- **Assegnato**: il task si trova nella $board_g$ "To Do";
- **In corso**: il task si trova nella $board_g$ "Doing";
- **Concluso**: il task si trova nella $board_g$ "Done";
- **Archiviato**: il task viene tolto dalle $board_g$.

In caso di problemi durante lo svolgimento del task, si procederà ad avvisare tempestivamente i membri del gruppo tramite $Telegram_g$.

4.2.3 Metriche

In questa sezione verranno esposte le metriche relative ai processi di pianificazione per l'applicazione Soldino. La classificazione di queste metriche viene esposta nella sezione 3.2.3 del documento corrente.

4.2.3.1 [MP01] Schedule Variance - SV

La Schedule Variance (SV) indica se un processo è in linea con la pianificazione designata, potendo stabilire se l'operazione è in anticipo o in ritardo.
Se SV > 0 significa che il progetto sta procedendo più velocemente rispetto a quanto pianificato. Viene calcolata utilizzando la seguente formula:

$$SV = BCWP - BCWS \tag{2}$$

Dove:

- **BCWP**: *Budgeted Cost of Work Performed*, è il valore (in giorni o €) delle attività realizzate alla data corrente. Rappresenta il valore prodotto dal progetto ossia il valore dei *deliverable*$_{\text{g}}$ rilasciati fino al momento della misurazione in seguito alle attività svolte;
- **BCWS**: *Budgeted Cost of Work Scheduled*, è il costo pianificato (in giorni o €) per realizzare le attività di progetto alla data corrente.

Per calcolare il BCWP occorre stimare il valore effettivamente maturato. Nei progetti in cui il costo del lavoro è nettamente predominante rispetto agli altri costi, spesso si approssima tale stima mediante tale procedimento:

$$BCWP_t = \sum BCWP_s \tag{3}$$

dove:

- **BCWP$_{\text{t}}$**: è l'indicatore BCWP totale del progetto;
- **BCWP$_{\text{s}}$**: è l'indicatore BCWP delle singole attività.

e:

$$BCWP_s = CA \times BAC_s \tag{4}$$

dove:

- **CA**: è l'indicatore del completamento attività;
- **BAC$_{\text{s}}$**: *Budget at Completion*, valore previsto per la realizzazione del progetto (valore iniziale previsto).

e:

$$CA = \frac{GD}{GD + GS} \tag{5}$$

dove:

- **GD**: equivale al numero di giorni erogati alla data;
- **GS**: equivale al numero di giorni stimati ancora da erogare.

4.2.3.2 [MP02] Cost Variance - CV

La Cost Variance (CV) permette di capire se in un determinato momento i costi effettivi siano maggiori o minori rispetto ai costi pianificati.
Se CV > 0 significa che il progetto produce con maggior efficienza rispetto a quanto stabilito.
Viene calcolata utilizzando la seguente formula:

$$CV = BCWP - ACWP \tag{6}$$

Dove:

- **BCWP**: *Budgeted Cost of Work Performed*, è il valore (in giorni o €) delle attività realizzate alla data corrente. Rappresenta il valore prodotto dal progetto ossia il valore dei $deliverable_g$ rilasciati fino al momento della misurazione in seguito alle attività svolte;
- **ACWP**: *Actual Cost of Work Performed*, è il costo effettivamente sostenuto (in giorni o €) alla data corrente.

Per calcolare l'ACWP occorre rilevare l'effort sostenuto ovvero l'impegno per risorsa per ciascuna attività.

4.2.3.3 [MP03] Rischi non previsti

Indica il numero di rischi che si sono presentati e che non sono stati individuati durante la fase di analisi dei rischi. Questa metrica viene misurata tramite un indice numerico che parte da 0.

4.2.3.4 [MP04] Efficacia della rimozione dei difetti

La metrica DRE (Defects Removal Effectiveness) può essere definita come la seguente percentuale:

$$DRE = \frac{DR}{DL} \times 100 \tag{7}$$

Dove DR è il numero di difetti rimossi e DL è il numero di difetti latenti.

Il numero di difetti latenti è dato dalla somma dei difetti rimossi e i difetti scoperti successivamente.

4.2.3.5 [MP05] Stage Team Advancement

Questa metrica indica il numero di task completati sui task totali di una fase di incremento, secondo la seguente formula:

$$STA = \frac{CT}{TT} \times 100 \tag{8}$$

4.2.3.6 [MP06] Percentuale requisti tracciati

Questa metrica indica il numero di requisiti tracciati sul numero di requisiti totali: un valore alto indica un'efficace attività di tracciamento dei requisiti. Si misura secondo la seguente formula:

$$PRA = \frac{TR}{TR} \times 100 \tag{9}$$

4.2.4 Strumenti

4.2.4.1 Diagrammi di Gantt

Lo strumento scelto per la creazione dei *diagrammi di Gantt*$_g$ è *GanttProject*$_g$.

4.2.4.2 Calcolo del consuntivo

Lo strumento usato per il calcolo del consuntivo è *Google Sheets*$_g$.

4.3 Formazione

Ogni membro del gruppo dovrà affinare la propria conoscenza sviluppando le competenze necessarie ad affrontare i compiti richiesti per il raggiungimento degli obiettivi del progetto. Per farlo può servirsi dei mezzi che ritiene più appropriati, dallo studio autonomo alla partecipazione a corsi formativi. A propria discrezione è possibile creare della documentazione informale per facilitare l'apprendimento ai propri colleghi.

4.3.1 Documentazione e guide per l'autoformazione

- Per l'utilizzo di LATEX:
  - https://www.latex-tutorial.com/tutorials/;
  - https://www.latex-project.org/help/documentation/#general-documentation.
- Per l'utilizzo di GitHub: https://guides.github.com/;
- Per l'utilizzo di Solidity: http://solidity.readthedocs.io;
- Per l'utilizzo di Metamask: https://metamask.io/;
- Per l'utilizzo di Ethereum: https://ethereumbuilders.gitbooks.io/guide/content/en/what_is_ethereum.html;
- Per l'utilizzo degli Ethereum token: https://blog.coinbase.com/a-beginners-guide-to-ether
- Per l'utilizzo di Truffle: https://truffleframework.com/docs/truffle/overview;
- Per l'utilizzo di Ganache: https://truffleframework.com/docs/ganache/overview;
- Per l'utilizzo di Raiden: https://raiden-network.readthedocs.io/en/stable/;
- Per l'utilizzo di Surge:
  - https://surge.sh/help/getting-started-with-surge;
  - https://surge.sh/help/.

- Per la Airbnb Javascript Style Guide: https://github.com/airbnb/javascript;
- Per la Airbnb React/JSX Style Guide: https://github.com/airbnb/javascript/tree/master/react;
- Per la Airbnb CSS/Sass Styleguide: https://github.com/airbnb/css.

A Standard di processo

A.1 [ISO/IEC 12207]

La norma ISO/IEC 12207 (International Organization for Standardization / International Electrotechnical Commission), dal titolo Systems and software engineering - Software life cycle processes, definisce i processi del ciclo di vita del software.

La norma rappresenta uno schema di riferimento comune per i processi relativi al ciclo di vita del software, comprensiva di una terminologia ben definita, che può essere utilizzata come riferimento dall'industria del software. La norma include i processi (process) con i rispettivi risultati e le attività (activity), suddivise in compiti (task), che devono essere svolte durante:

- L'acquisizione di sistemi software;
- La creazione di prodotti software a sé stanti o di servizi software;
- La fornitura, lo sviluppo, la gestione operativa e la manutenzione di prodotti software quando siano svolte sia internamente che esternamente ad una organizzazione.

La norma fornisce quindi un modello per la definizione, il controllo e il miglioramento dei processi relativi al ciclo di vita del software.

Lo standard 12207 nella revisione del 2017 definisce 30 diversi processi nel ciclo di vita del software raggruppandoli in 4 macro-categorie:

- Agreement Processes;
- Organizational project-Enabling Processes;
- Technical Management Processes;
- Technical Processes.

Le indicazioni contenute nella norma [ISO/IEC 12207] possono essere efficacemente integrate con quelle della norma [ISO/IEC 15504].

[18] https://en.wikipedia.org/wiki/ISO/IEC_12207

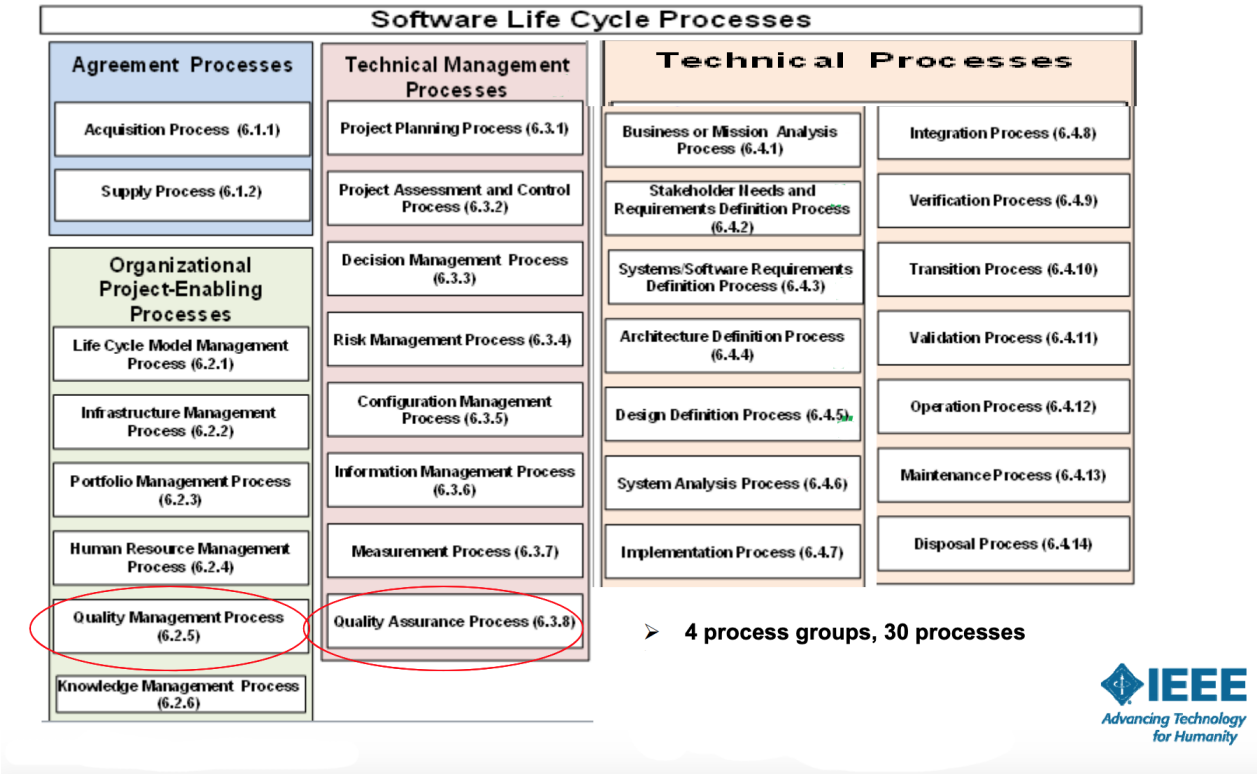

Figura 18:]
Standard di processo - [ISO/IEC 12207:2017] [18]

A.2 [ISO/IEC 15504]

Lo standard [ISO_g/IEC_g 15504] - Software Process Improvement and Capability dEtermination, o $SPICE_g$, nasce dall'esigenza di uniformare le metodologie per il miglioramento dei processi di sviluppo del software.

La $suite_g$ di documenti fornisce un modello di riferimento internazionale per la valutazione dei processi.

Il modello $SPICE_g$ suddivide i processi in cinque diverse categorie:

- Customer-supplier;
- Engineering;
- Supporting;
- Management;
- Organization.

[19] https://en.wikipedia.org/wiki/ISO/IEC_15504

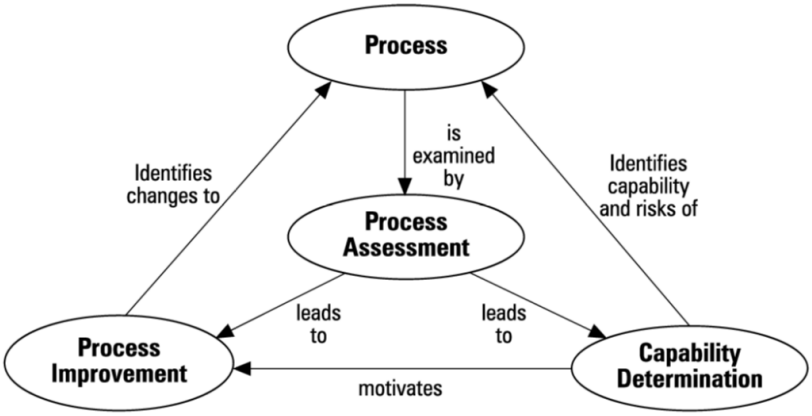

Figura 19:]
Assessment di processo - [ISO/IEC 15504][19]

Per ciascun processo all'interno di queste categorie si definiscono sei possibili livelli di maturità, dotati di attributi utili alla loro valutazione:

Livello	Nome	Descrizione
5	Optimizing process	Il processo cambia e si adatta dinamicamente per raggiungere gli obiettivi aziendali
4	Predictable process	Il processo viene eseguito costantemente entro limiti definiti per raggiungere i risultati attesi
3	Established process	Il processo viene eseguito basandosi su principi dell'ingegneria del software ed è in grado di raggiungere i risultati fissati
2	Managed process	Il processo viene eseguito in modo controllato in base a obiettivi definiti
1	Performed process	Il processo attuato raggiunge i suoi risultati
0	Incomplete process	Il processo non viene attuato o non riesce a raggiungere i suoi risultati

Tabella 2: Livello di maturità dei processi

Per ogni livello esistono specifici attributi atti a misurare la *capability*$_g$ del processo in esame:

- **Optimizing process**:
  - **Process innovation**: i cambiamenti strutturali, di gestione e di esecuzione vengono gestiti in modo controllato per raggiungere i risultati fissati;
  - **Process optimization**: le modifiche al processo sono identificate e implementate per garantire il miglioramento continuo nella realizzazione degli obiettivi di business dell'organizzazione.

  - **Process definition**: l'esecuzione del processo si basa su standard di processo per raggiungere i propri obiettivi;
  - **Process deployment**: capacità del processo di attingere a risorse tecniche e umane appropriate per essere attuato efficacemente.
- **Managed process**:
  - **Performance management**: capacità del processo di elaborare un prodotto coerente con gli obiettivi fissati;
  - **Work product management**: capacità del processo di elaborare un prodotto documentato, controllato e verificato.
- **Performed process**:
  - **Process performance**: capacità di un processo di raggiungere gli obiettivi trasformando input identificabili in output identificabili.

Ognuno dei sopracitati attributi viene valutato secondo la seguente scala di valori:

- Not achieved (0 - 15%);
- Partially achieved (>15% - 50%);
- Largely achieved (>50%- 85%);
- Fully achieved (>85% - 100%).

B Standard di qualità di prodotto

B.1 [ISO/IEC 25010]

Lo standard [ISO_g/IEC_g 25010] è parte di un più ampio insieme di standard ovvero la famiglia [25000], detta anche SQuaRE - System and Software Quality Requirements and Evaluation.

La norma [25010] del 2011 dal titolo "Systems and software engineering – SQuaRE – System and software quality models" fornisce il nuovo modello di *qualità*$_g$ che rimpiazza quello definito dalla norma [9126-1].

Il nuovo standard, pur recependo dal precedente la differenza tra *qualità*$_g$ interna ed esterna con lo stesso significato, definisce un unico Modello integrato, detto della *qualità*$_g$ del Prodotto. Questo intende raccogliere tutte le proprietà dinamiche e statiche, quindi indistintamente quelle interne ed esterne.

I nuovi standard della famiglia SQuaRE, infatti, hanno come scopo quello di allargare il proprio ambito di applicabilità, generalizzando i principi dell'Ingegneria del Software anche ad altri domìni.

Come per lo standard [ISO_g/IEC_g 9126], viene proposto un Modello della *qualità*$_g$ in Uso, ancora importante per poter rappresentare la percezione che l'utente ha del prodotto.

Il Modello della *qualità*$_g$ in Uso [*ISO*$_g$/*IEC*$_g$ 25010] viene rivoluzionato rispetto al vecchio standard, con l'introduzione di un secondo livello di sottocaratteristiche, a voler modellare in maniera più ampia la *qualità*$_g$ percepita.

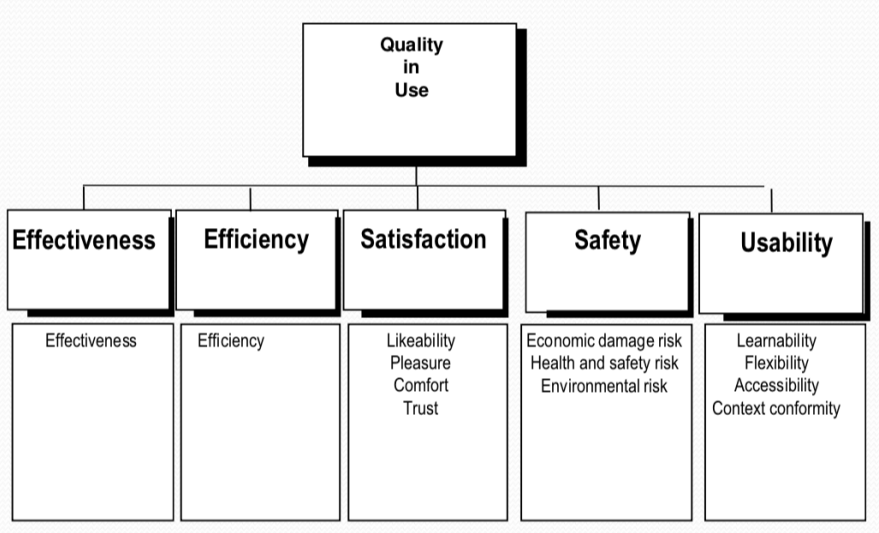

Figura 20:]
Qualità in Uso - [ISO/IEC 25010][20]

- **Effectiveness**: misura accuratezza e completezza con la quale gli utenti raggiungono gli obiettivi specificati;
- **Efficiency**: misura le risorse spese in relazione all'accuratezza e completezza con la quale gli utenti raggiungono gli obiettivi;
- **Satisfaction**: misura quanto le esigenze degli utenti risultano soddisfatte durante l'utilizzo di un software in uno specifico contesto d'uso. Questa caratteristica presenta i seguenti attributi:
  - **Likeability**: misura il grado di soddisfazione percepito da un utente per il raggiungimento degli obiettivi pragmatici, compresi i risultati d'uso e le conseguenze dell'uso;
  - **Trust**: misura quanto un utente o un altro *stakeholder*$_g$ confidi che il software si comporti come previsto.
  - **Pleasure**: misura quanto un utente ottiene piacere dal soddisfacimento delle proprie esigenze;
  - **Comfort**: misura quanto un utente è soddisfatto dal comfort fisico.
- **Safety**: misura quanto un software è in grado di ridurre i potenziali rischi economici, per la salute e l'ambiente. Questa caratteristica presenta i seguenti attributi:
  - **Economic damage risk**: misura quanto un software è in grado di ridurre il potenziale rischio economico legato alla situazione finanziaria, al patrimonio immobiliare, all'*efficienza*$_g$, alla reputazione o altre risorse nei contesti d'uso previsti;

[20] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

  - **Health and safety risk**: misura quanto un software è in grado di ridurre il potenziale rischio per le persone nei contesti d'uso previsti;
  - **Environmental risk**: misura quanto un software è in grado di ridurre il potenziale rischio per la proprietà o per l'ambiente nei contesti d'uso previsti.

- **Usability**: grado con cui un prodotto o un sistema può essere utilizzato dall'utente per raggiungere gli obiettivi specificati con *efficacia*$_g$, *efficienza*$_g$ e soddisfazione in un determinato contesto di utilizzo. Questa caratteristica presenta i seguenti attributi:
  - **Learnability**: è la capacità di un prodotto software di essere appreso facilmente;
  - **Accessibility**: è il grado con cui un prodotto software può essere utilizzato da persone con la più ampia gamma di caratteristiche e capacità;
  - **Context conformity**: è il grado con cui un prodotto software risulti conforme al contesto nel quale viene eseguito;
  - **Flexibility**: misura quanto un software può essere utilizzato con *efficacia*$_g$, *efficienza*$_g$, assenza di rischio e soddisfazione in contesti che vanno oltre quelli inizialmente indicati nei requisiti, come l'adattabilità a nuove tipologie di utenti, a nuove funzionalità, attività o ambienti operativi.

Il Modello della *qualità*$_g$ del Prodotto [*ISO*$_g$/*IEC*$_g$ 25010] è costituito da otto caratteristiche di I livello e trentuno sottocaratteristiche di II livello, direttamente o indirettamente misurabili attraverso le metriche interne ed esterne, presentate nella norma 25023.

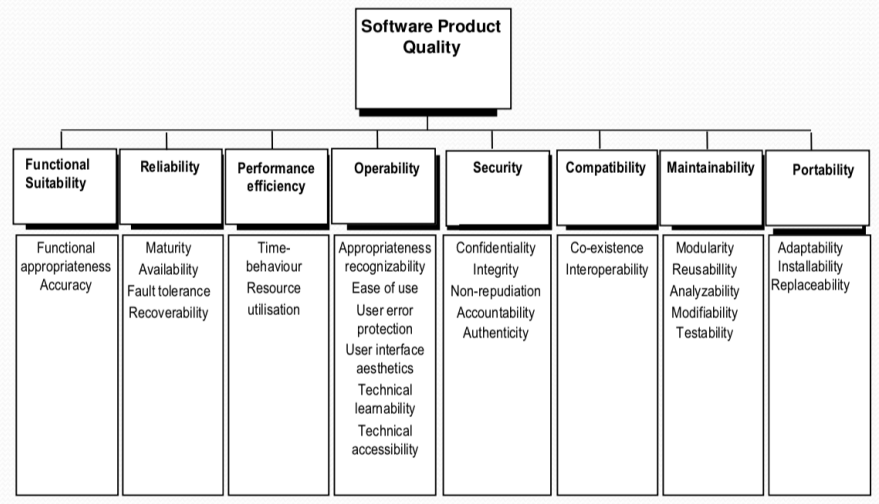

Figura 21:]
Qualità del Prodotto - [ISO/IEC 25010][21]

Le 8 caratteristiche sono:

- **Functional Suitability**: questa caratteristica rappresenta il grado con cui un prodotto o un sistema fornisce funzioni che soddisfano esigenze, dichiarate e implicite, quando

[21] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

vengono utilizzate in condizioni specificate. Questa caratteristica è composta dalle seguenti sottocaratteristiche:

  - **Functional completeness**: grado con cui l'insieme di funzioni copre tutte le attività e gli obiettivi specificati dall'utente;
  - **Functional correctness**: grado con cui un prodotto o un sistema fornisce i risultati corretti con il grado di precisione necessario;
  - **Functional appropriateness**: grado con cui le funzioni facilitano il raggiungimento di determinati compiti e obiettivi.

- **Performance efficiency**: questa caratteristica rappresenta la performance relativa alla quantità di risorse utilizzate in determinate condizioni. Questa caratteristica è composta dalle seguenti sottocaratteristiche:
  - **Time behaviour**: grado con cui i tempi di risposta, di elaborazione e le velocità di *throughput*$_{\text{g}}$ di un prodotto o di un sistema, durante l'esecuzione delle sue funzioni, soddisfano i requisiti;
  - **Resource utilization**: grado con cui le quantità e i tipi di risorse utilizzate da un prodotto o da un sistema soddisfano i requisiti;
  - **Capacity**: grado con cui i limiti massimi di un prodotto o di un parametro di sistema soddisfano i requisiti.
- **Compatibility**: grado con cui un prodotto, sistema o *componente*$_{\text{g}}$ può scambiare informazioni con altri prodotti, sistemi o *componenti*$_{\text{g}}$ e/o svolgere le funzioni richieste, condividendo lo stesso ambiente hardware o software. Questa caratteristica è composta dalle seguenti sottocaratteristiche:
  - **Co-existence**: grado con cui un prodotto può svolgere le funzioni richieste in modo efficiente condividendo un ambiente e risorse comuni con altri prodotti, senza alcun impatto negativo verso altri prodotti;
  - **Interoperability**: grado con cui due o più sistemi, prodotti o *componenti*$_{\text{g}}$ possono scambiarsi informazioni e utilizzare le informazioni scambiate.
- **Usability**: grado con cui un prodotto o un sistema può essere utilizzato da utenti specifici per raggiungere gli obiettivi desiderati con *efficacia*$_{\text{g}}$, *efficienza*$_{\text{g}}$ e soddisfazione in un determinato contesto di utilizzo. Questa caratteristica è composta dalle seguenti sottocaratteristiche:
  - **Appropriateness recognizability**: grado con cui gli utenti possono riconoscere se un prodotto o un sistema è adatto alle loro esigenze;
  - **Learnability**: grado con cui un prodotto o un sistema può essere utilizzato per raggiungere gli obiettivi di apprendimento, utilizzare il prodotto o il sistema con *efficacia*$_{\text{g}}$, *efficienza*$_{\text{g}}$, assenza di rischio e soddisfazione in un contesto di utilizzo specifico;
  - **Operability**: grado con cui un prodotto o un sistema possiede attributi che lo rendono facile da utilizzare e controllare;

  - **User error protection**: grado con cui un sistema protegge gli utenti dal commettere errori;
  - **User interface aesthetics**: grado con cui un'*interfaccia utente*$_g$ consente un'interazione piacevole e soddisfacente per lo stesso;
  - **Accessibility**: grado con cui un prodotto o un sistema può essere utilizzato da persone con la più ampia gamma di caratteristiche e capacità per raggiungere un obiettivo specificato.

- **Reliability**: grado con cui un sistema, un prodotto o un *componente*$_g$ esegue determinate funzioni in condizioni e tempo specificati. Questa caratteristica è composta dalle seguenti sottocaratteristiche:
  - **Maturity**: grado con cui un sistema, prodotto o *componente*$_g$ soddisfa le esigenze di affidabilità durante il normale funzionamento;
  - **Availability**: grado con cui un sistema, prodotto o *componente*$_g$ è operativo e accessibile quando richiesto per l'uso;
  - **Fault tolerance**: grado con cui un sistema, un prodotto o un *componente*$_g$ funziona come previsto nonostante la presenza di errori hardware o software;
  - **Recoverability**: grado con cui, in caso di interruzione o guasto, un prodotto o un sistema può recuperare i dati interessati e ristabilire lo stato desiderato del sistema.

- **Security**: grado con cui un prodotto o un sistema protegge informazioni e dati dando le corrette autorizzazioni a persone, prodotti o sistemi con il grado di accesso appropriato. Questa caratteristica è composta dalle seguenti sottocaratteristiche:
  - **Confidentiality**: grado con cui un prodotto o un sistema garantisce che i dati siano accessibili solo alle entità autorizzate ad averne accesso;
  - **Integrity**: grado con cui un sistema, un prodotto o un *componente*$_g$ impedisce l'accesso non autorizzato o la modifica di programmi o dati informatici;
  - **Non-repudiation**: grado con cui è possibile dimostrare che azioni o eventi hanno avuto luogo, al fine di non essere ripudiati successivamente;
  - **Accountability**: grado con cui le azioni di un'entità possono essere tracciate in modo univoco;
  - **Authenticity**: grado con cui l'identità di un soggetto o di una risorsa può essere dimostrata come quella rivendicata.

- **Maintainability**: questa caratteristica rappresenta il grado di *efficacia*$_g$ ed *efficienza*$_g$ con cui un prodotto o un sistema può essere modificato per migliorarlo, correggerlo o adattarlo ai cambiamenti dell'ambiente e a nuovi requisiti. Questa caratteristica è composta dalle seguenti sottocaratteristiche:

  - **Modularity**: grado con cui un sistema o un programma è composto da *componenti*$_{\text{g}}$ discreti, in modo tale che la modifica di una parte abbia un impatto minimo sull'interezza del sistema;
  - **Reusability**: grado con cui un bene può essere utilizzato in più di un sistema o nella costruzione di altre attività;
  - **Analysability**: grado di *efficacia*$_{\text{g}}$ ed *efficienza*$_{\text{g}}$ con il quale è possibile valutare l'impatto su un prodotto o un sistema di un cambiamento previsto per una o più parti. Un buon grado di Analysability risulta utile per diagnosticare le carenze o cause di guasti in un prodotto;
  - **Modifiability**: grado con cui un prodotto o un sistema può essere modificato in modo efficace ed efficiente senza introdurre difetti o degradare la *qualità*$_{\text{g}}$ del prodotto esistente;
  - **Testability**: grado di *efficacia*$_{gg}$ ed *efficienza*$_{\text{g}}$ con cui possono essere stabiliti i criteri di test per un sistema, prodotto o *componente*$_{\text{g}}$.
- **Portability**: grado di *efficacia*$_{\text{g}}$ ed *efficienza*$_{\text{g}}$ con cui un sistema, prodotto o *componente*$_{\text{g}}$ può essere trasferito da un hardware, software o altro ambiente operativo a un altro. Questa caratteristica è composta dalle seguenti sottocaratteristiche:
  - **Adaptability**: grado con cui un prodotto o un sistema può essere adattato in modo efficace ed efficiente ad hardware, software o altri ambienti operativi, di utilizzi diversi o in evoluzione;
  - **Installability**: grado di *efficacia*$_{\text{g}}$ ed *efficienza*$_{\text{g}}$ con cui un prodotto o un sistema può essere installato e/o disinstallato con successo in un determinato ambiente;
  - **Replaceability**: grado con cui un prodotto può sostituire un altro prodotto software specificato per lo stesso scopo nello stesso ambiente.

C Ciclo di Deming

Al fine di garantire la *qualità*$_{\text{g}}$ dei processi è necessaria un'organizzazione e una strategia focalizzata al miglioramento continuo. Il *ciclo di Deming*$_{\text{g}}$ o *PDCA*$_{\text{g}}$, da noi adottato, è stato formalizzato per affrontare in maniera rigorosa e sistematica qualsiasi attività assicurando il non decremento della *qualità*$_{\text{g}}$.

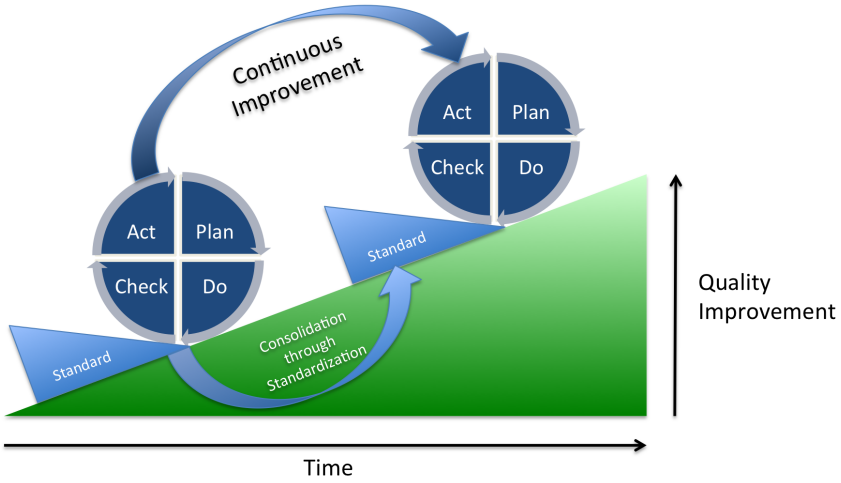

Figura 22: Ciclo di Deming[22]

Il $PDCA_g$ è un metodo iterativo suddiviso in quattro fasi:

- **P - Plan**: identificare e descrivere il problema analizzando i suoi aspetti principali. Per fornire una descrizione accurata del problema è necessario raccogliere dei dati tramite osservazione e analisi. Successivamente vengono definiti gli obiettivi di massima e identificati i benefici che ne derivano. In seguito all'identificazione delle cause del problema e degli ostacoli, vengono determinati gli interventi da attuare per rimuoverli e i risultati attesi dall'operazione. Vengono pianificate le azioni da svolgere con la relativa allocazione delle risorse necessarie. Infine avviene la determinazione delle metriche per misurare i miglioramenti o gli scostamenti da quanto previsto;
- **D - Do**: esecuzione di quanto pianificato, dapprima in contesti circoscritti. È la fase operativa del ciclo dove viene attuato il piano, eseguito il processo, oppure creato il prodotto. In questa fase vengono inoltre raccolti i dati per la creazione di materiale da analizzare successivamente nelle fasi di "Check" e "Act";
- **C - Check**: test e controllo, studio dei risultati e dei riscontri. In questa fase è necessario studiare i risultati misurati e raccolti nella fase "Do" confrontandoli con i risultati attesi, gli obiettivi del "Plan", per verificarne le eventuali differenze;
- **A - Act**: azione per rendere definitivo e/o migliorare il processo estendendo quanto testato precedentemente in contesti circoscritti all'intera organizzazione. Richiede azioni correttive sulle differenze significative tra i risultati effettivi e previsti. Analizza le differenze per determinarne le cause e dove applicare le modifiche per ottenere il miglioramento del processo o del prodotto.

Al completamento delle quattro fasi, il miglioramento viene standardizzato. Quando un procedimento, attraverso questi quattro passaggi, non comporta la necessità di migliorare la portata a cui è applicato, il ciclo $PDCA_g$ può essere raffinato per pianificare e migliorare con maggiore dettaglio la successiva iterazione.

[22] https://en.wikipedia.org/wiki/PDCA

D Tipologie di test

Per ottenere un buon risultato durante la fase di sviluppo, è necessario l'ausilio di test per verificare il corretto funzionamento del software.
Questi test si suddividono in cinque tipologie:

- Test di unità;
- Test di integrazione;
- Test di sistema;
- Test di validazione;
- Test di regressione.

Poiché il programma può essere eseguito in contesti differenti, è necessario eseguire questi test per trovare ogni possibile *criticità*$_g$, è dunque fondamentale ripetere le prove e confrontare i risultati ottenuti con quelli attesi.

D.1 Test di unità

Per test di unità si intende l'attività di testing, prova o collaudo di singole unità software. Per unità si intende normalmente il minimo *componente*$_g$ di un programma dotato di funzionamento autonomo; a seconda del paradigma di programmazione o linguaggio di programmazione, questo può corrispondere per esempio a una singola funzione nella programmazione procedurale, o una singola classe o un singolo metodo nella programmazione a oggetti.
In questa fase del progetto non sono stati progettati test di unità, in quanto le funzioni e i metodi che il gruppo ha implementato per il *Proof of Concept*$_g$, seppur seguendo il modello incrementale, potrebbero non essere presenti nel prodotto finale.

Questa tipologia di test viene eseguita tramite l'ausilio delle *componenti*$_g$:

- **Stub**: porzione di codice utilizzata in sostituzione di altre funzionalità software in quanto può simulare il comportamento di codice esistente o l'interfaccia COM, e temporaneo sostituto di codice ancora da sviluppare;
- **Logger**: *componente*$_g$ non intrusivo di registrazione dei dati di esecuzione per l'analisi dei risultati;
- **Driver**: *componente*$_g$ che simula il comportamento di unità di più alto livello che non sono ancora state sviluppate. I driver fungono da sostituti temporanei delle unità e agiscono come farebbero nel prodotto finale.

Attraverso questi test si vuole collaudare ogni possibile unità sviluppata del progetto software tramite:

- **Test funzionale**: dato un insieme di dati in input, si ottiene un risultato che deve essere corrispondente a quello atteso;
- **Test strutturale**: detto anche white box o verifica strutturale, è un particolare tipo di test che viene effettuato per rilevare errori in uno o più *componenti*$_g$ di un sistema software. Il suo funzionamento si basa su alcuni criteri che hanno lo scopo di trovare

dati di test che consentano di percorrere tutto il programma. Per trovare un errore nel codice, infatti, bisogna usare dei dati che "percorrono" la parte erronea del programma. Se eseguito correttamente e senza particolari eccezioni, il test può coprire fino al 90% delle istruzioni.

D.2 Test di integrazione

Questi test mirano a verificare la corretta integrazione tra le varie unità logiche che formano il software. I problemi rilevati da questi test possono far emergere una scarsa attenzione alla *progettazione*$_g$ o una scarsa *qualità*$_g$ dei test di unità.
In questa fase vengono trovati i problemi di interfaccia tra i diversi blocchi di codice, accertandosi che siano conformi alle specifiche e che i flussi di controllo siano stati provati.

D.3 Test di sistema

Questi test solitamente si basano sulle funzionalità espresse nelle specifiche e nei requisiti dell'applicazione e possono coinvolgere diverse configurazioni software e hardware. Anche se l'approccio utilizzato è per lo più di tipo black box testing, quindi orientato ai test funzionali, vengono controllate anche altre caratteristiche strutturali come sicurezza, usabilità e *manutenibilità*$_g$. Un classico test di sistema, oltre a quelli legati alla sicurezza, è il cosiddetto "stress test", che ha lo scopo di destabilizzare il sistema ponendolo in uno stato di stress, quindi con un sovraccarico o una sottrazione di risorse.

D.4 Test di validazione

Il test di validazione è il controllo finale e definitivo, tale accertamento deve assicurare che il prodotto rispetti i requisiti forniti dal *Committente*$_g$. Questa procedura viene solitamente effettuata in presenza del cliente o di chi ne fa le veci ed è importante che il collaudo sia condotto in condizioni operative reali.

D.5 Test di regressione

I test di regressione consistono nel rieseguire in modo selettivo i test di unità e di integrazione. Questa attività viene eseguita ogni volta che viene effettuata qualche modifica nel codice, come ad esempio l'inserimento di nuove funzionalità o la correzione di un *bug*$_g$. Tali variazioni nel codice comportano l'esecuzione di tutti i test elencati poc'anzi. Questa tipologia di test risulta essere essenziale, poiché l'introduzione di modifiche all'interno del codice molto spesso tende a creare ulteriori difetti.